FACULTY OF COMPUTING SCIENCES AND ENGINEERING

COMPONENT-BASED CONTROL SYSTEM DEVELOPMENT FOR AGILE MANUFACTURING MACHINE SYSTEMS

XI CHEN

A dissertation submitted in partial fulfilment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

JULY 2003

TEXT BOUND CLOSE TO THE SPINE IN THE ORIGINAL THESIS

Declaration

No part of the material described in this thesis has been submitted for the award of any other degree or qualification in this or any other university or college of advanced education.

Acknowledgement

I would like to express my sincere gratitude to Dr. Junsheng Pu and Professor Philip Moore for their patient guidance and continual encouragement throughout my research work. Furthermore, I would like to thank them for offering me the studentship to carry out this research work.

I also want to thank Mr. C. B. Wong, Dr. S. K. Chong, Mr. J. Adolfsson, Mr. Amos Ng, Dr. L. G. Shen, and the whole VIR-ENG team for their support and collaboration in this research project. Special thanks go to Dr. C. Xie, Dr. Z Zhang, Mr. D Zhang, Mr. X. Li, and Dr. H Dai, and all those who give me help directly or indirectly in completing this research work.

Finally, my special thanks also go to Mr. H. Klas and Mr. Bengts from Euromation (Volvo automation) in Sweden, who helped greatly in particular with their invaluable practical advice and comments for this work in building the control system for the demonstration cell.

Synopsis

It is now a common sense that manufactures including machine suppliers and system integrators of the 21st century will need to compete on global marketplaces, which are frequently shifting and fragmenting, with new technologies continuously emerging. Future production machines and manufacturing systems need to offer the "agility" required in providing responsiveness to product changes and the ability to reconfigure. The primary aim for this research is to advance studies in machine control system design, in the context of the European project VIR-ENG – "Integrated Design, Simulation and Distributed Control of Agile Modular Machinery".

A component-based paradigm has been adopted as the general framework for the design and implementation of agile modular manufacturing machines and their associated control systems. Thus, the entire control system is viewed as a set of components, which are connected through an underlying communication mechanism. Furthermore, since the component concept can be adopted for the design of mechanical system (modular), electronics system, and information infrastructure system, a component-based development approach can be used to streamline the various design stages with the view to devise a seamless integrated development process and support environment. This research work is largely concerned with the detailed design of the control architecture and prototype implementation of tools and methods for a working demonstrator at an industrial site, Euromation (Volvo) in Sweden.

A method named Component Responsibility and Collaboration (CRC) has been proposed for the control architecture design, as a guide for the subsequent development process, which involves both hardware and software components. By adopting the Unified Modelling Language (UML) as the modelling language, CRC provides a methodology and some general principles to assist the design activities. To facilitate the use of this method, a supporting tool was developed to automate the associated clerical tasks.

IEC 1131-3 languages have been adopted in this research project for the representation and implementation of the control logic components, which is seemingly the only widely recognised standard used in industry for control systems. A key contribution of this research is an extension of underlying mechanisms, which can be used to facilitate the adoption of IEC 1131-3 to support component-based development. Bearing this in mind, a suite of mechanisms and tools have been developed, which include:

- Connector, a software mechanism, which provides the underlying component operation mechanisms, such as changing the component property, invoking the component method and composing the components. This allows the use of IEC 1131-3 languages in collecting the component.
- VCon, a utility program, which automates most of the connector operations to reduce the level of complexity in implementation.
- Adapter, a software mechanism, which enables the IEC1131-3 languages (and other languages) to be used in building the component. It provides a better way for development the code rather than by "Copy" and "Paste" of the source code in the traditional method.
- CBuilder, a utility program, which eases the process of applying the adapter by means of a graphical user interface.

It should be recognised that the control system design environment is not just an isolated entity, but integral with the mechanical design and run-time support design environments. The design methods and tools thus developed have been used in building a demonstrator cell at Euromation (a Volvo group company in Sweden), incorporating typical assembly machinery types (e.g. assembly machine, modular conveyor, AGV, etc.). The methods and tools described in this thesis have been successfully used in carrying out several real production scenarios. These include (a) the design and implementation of a new machine system, (b) new product introduction, and (c) reconfiguration of existing machine systems.

Table of Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

AGV	Automatic Guided Vehicle
AMEF	Agile Manufacturing Enterprise Forum
AMS	Agile Manufacturing System
API	Application Programming Interface
CAD	Computer-Aided Design
CADE	Control Architecture Design Environment (VIR-ENG)
CAE	Computer-Aided Engineering
CASE	Computer-Aided System Environment
CBuilder	Component Builder
CDE	Component Design Environment
CIM	Computer Integrated Manufacturing
CLPE	Control Logic Programming Environment (VIR-ENG)
CNC	Computer Numerical Control
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CRC	Component Responsibility and Collaboration
CSDE	Control System Design Environment (VIR-ENG)
DCOM	Distributed Component Object Model
DCSE	Distributed Control System Environment (VIR-ENG)
DRE	Distributed Run-time Environment (VIR-ENG)
ESPIRT	The European Union Information Technologies Programme
FB	Function Block (IEC 1131-3)
FBD	Function Block Diagram
FMS	Flexible Manufacturing System
HMI	Human Machine Interface
IDEF	Integration Definition for Function Modelling
IEC	International Electrotechnical Commission
IIS	Infrastructure and Integration Services (VIR-ENG)
IL	Instruction List

I/O	Input/Output
ISO	International Standard Organisation
LD	Ladder Diagram
LON	Local Operating Network (Echelon)
MMDE	Modular Machine Design Environment (VIR-ENG)
NC	Numerical Control
OMAC	Open Modular Architecture Controller
OMG	Object Management Group
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OPC	OLE for Process Control
ORB	Object Request Broker
OSACA	Open System Architecture for Controls within Automation systems
PLC	Programmable Logic Controller
POU	Program Organisation Unit (IEC 1131-3)
QUEST	QUeuing Event Simulation Tool
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SCADA	Supervisory Control And Data Acquisition
SDL	Specification and Description Language
SDS	Smart Distributed System (Honeywell)
SFC	Sequential Function Chart (IEC 1131-3)
SME	Small-to- Medium Enterprise
ST	Structure Text (IEC 1131-3)
UMC	Universal Machine Controller
UML	Unified Modelling Language
VCon	VECOM CONfiguring
VECOM	VIR-ENG COMponent
VIR-ENG	Virtual-Engineering - ESPIRT Framework IV project 'Integrated Design, Simulation, and Distributed Control of Agile Modular Manufacturing Machinery'
XML	eXtensible Mark-up Language

Chapter 1 Introduction

1.1 Background

The agile manufacturing paradigm was formulated in the early 1990's in response to the dynamics in the 'new economy' resulting from global competitiveness (Clark et al., 1991; Womack et al., 1990; Termini, 1996). In this context, the widely accepted definition of 'agility' refers to the capability to manufacture products to meet frequent changes in customer demands, in a timely, flexible and cost-effective manner. This research adopts the definition of 'agile' machine system in the context of product assembly for a production line, that is *machine systems that are designed and built with the inherent capability to accomplish rapid product changeover with model and/or feature variants*. For custom-built machine systems, it is an identifiable trend to demand manufacturing services (rather than production units) that could be rented only for the term of use. This is particularly true for SMEs, because they do not have great financial resources. In response to the need of agile machine systems, in particular customer-driven machine system design and building (often one-off), the growing tendency is that the role of machine builders is extended to become one of service providers. Therefore, for machine builders or "service providers", agility is required in both the development phase (short lead time) as well as the use phase (rapid reconfiguration and maintenance, which is associated with high reliability).

One of the most important requirements leading to implementation of the agile machine system is to provide a machine control system that is able to respond and adapt to changing production environment. Following the argument to build agile machine systems, this research seeks to develop the context of "agile" control system in terms of concepts and solutions.

A significant body of research worldwide addresses building machine control system solutions in multi-vendor based systems, sometimes referred to as 'open-architecture' control. For many, open control and agile control sound similar, but open control systems do not necessarily imply flexibility and re-configurability, which are features of agile control systems. Moreover, most of these programmes focus on the higher-

level system architecture and do not specifically address the issues pertinent to device level components.

From the definition of agile control systems, the following aspects need to be considered:

1. *Distributed versus Centralised.* The complex nature of today's manufacturing systems makes them difficult to control in a centralised manner. Distributed control systems offer the potential to reduce the risk of whole system failure and increase system flexibility. Furthermore, this is considered to be an essential enabler for the agile enterprise, where it forms a flexible and adaptable operational architecture to respond to the technical changes of structure (Weston, 1999). Practically, during the process to realise manufacturing systems, the associated control system has to be decomposed, replacing one big problem with many small ones. Each subsystem is only concerned with finite functionalities.
2. *Configurability and Reconfigurability.* Control agility should encompass the ability to cope with product variety and process variety. In detail, it should be possible to introduce new models or variants to the manufacturing system without the need to redesign the production facilities.
3. *Some level of autonomy.* The control system should consist of distributed, autonomous control units able to temporarily operate independently. Upper and lower level systems should continue to operate and maintain production if adjacent levels of control fail.
4. *A degree of Openness.* Reconfigurability and interchangeability require architectures that are flexible and that support tools from a variety of sources and domains. This forces a shift away from traditional control system implementation to open architectures, which is one of the directions for solving this problem. The use of standardised interfaces and network protocols combined should help to deliver more 'open systems'.
5. *Modularity.* Modularity combined with autonomy of operation and the open system architecture provides increased extensibility and ease of reconfiguration. Modularity also allows the implementation and development of control systems with reasonable effort and helps faster 're-use' of solutions.
6. *Provision for integration.* Although distributed control systems are required, a supporting integration infrastructure is required to realise a coherent architecture,

able to support the ultimate goal of the manufacturing system as well as to allow integration with the other enterprise applications.

To design and implement these features in an agile control system can be a time-consuming, demanding and tedious task, and to cope with the changing demands is another challenge. For engineers, systematic design methods and associated tools are highly desired. The key challenges and problem areas have been identified as follows:

1. Machine building can be a very complex process, typically involving a team of engineers with different skills, e.g. mechanical engineers, electronics engineers, control engineers and software engineers, together with a project manager. However, in the light of current practice in industry, the control system development is isolated from the other aspects of building the complete machine system, often relying purely on human interactions.
2. Current approaches to the design and building of machine control systems are characterised by minimal software re-use and poor verification of the customer's requirements. Subsequently, they limit confidence in the correctness of proposed designs and their associated costs, and have time consuming and unpredictable commissioning phases. Furthermore, they are also costly in terms of system maintenance and enhancement.
3. Until now, the development of contemporary machine control systems has largely been driven by hardware considerations and constraints. With the shift from hardware-oriented to software-oriented system, there are insufficient software elements, which exhibit reusability and adaptability that can be used as building blocks for machine control systems.
4. A standardised infrastructure is necessary to enable system users and system integrators to purchase and replace parts of the system without adversely affecting the rest of the system or requiring extended integration effort. It would imply the requirements for the formal description of communication interfaces and the functional capabilities of the modules, which is essential information for system users and integrators.
5. A simulation-based approach is very useful for control logic design and prototyping. However, conventional simulation-based design generally uses a separate simulator to perform the simulation process (Zobel and Lee, 1992). This

implies that the development of control systems is isolated from simulation and benefits little from simulation.

6. In machine system design, efforts have been directed to the creation of standards and to the formulation of guidelines. However, few efforts have focused on the development of tools that make it possible for several designers to interact concurrently through a workflow management system.

1.2 Research Areas and Relationship with VIR-ENG

This Ph.D. research work is carried out in the context of the Framework IV EC project: VIR-ENG ("Integrated Design, Simulation and Distributed Control of Agile Modular Manufacturing Machinery"), which is the integrated environment to facilitate the entire lifecycle support for machine system design, implementation and operation and to readily deliver machine systems that are flexible, cost-effective and which offer proven solutions.

In the context of VIR-ENG, the process to realise agile modular manufacturing systems can be depicted as in figure 1.1, which starts with initial machine requirements from the customer, through co-operative multi-disciplinary design and implementation and finally delivers the machine system. In terms of design environment, the VIR-ENG environment is composed of three primary design elements: mechanical system design (blue area), control system design (orange area), and run-time support system design (green area). These design elements are coherently intertwined together, as an integral environment as depicted by the blended colour areas. These intersections indicate integration issues.

With regard to this research work, the main research area is to design and realise machine control systems, which is represented as the orange colour area. The research also addresses the integration issues, which are indicated by the blended colour areas. However, the electrical and electronic hardware design is outside the scope of this research work, which is indicated by a white colour. In detail, this work provides a range of methodologies and supporting tools to enable the control system development process within the entire process, which covers the requirement analysis,

control architecture design, control system design, and control system implementation. It also facilitates the re-configuration and re-design of existing control systems.

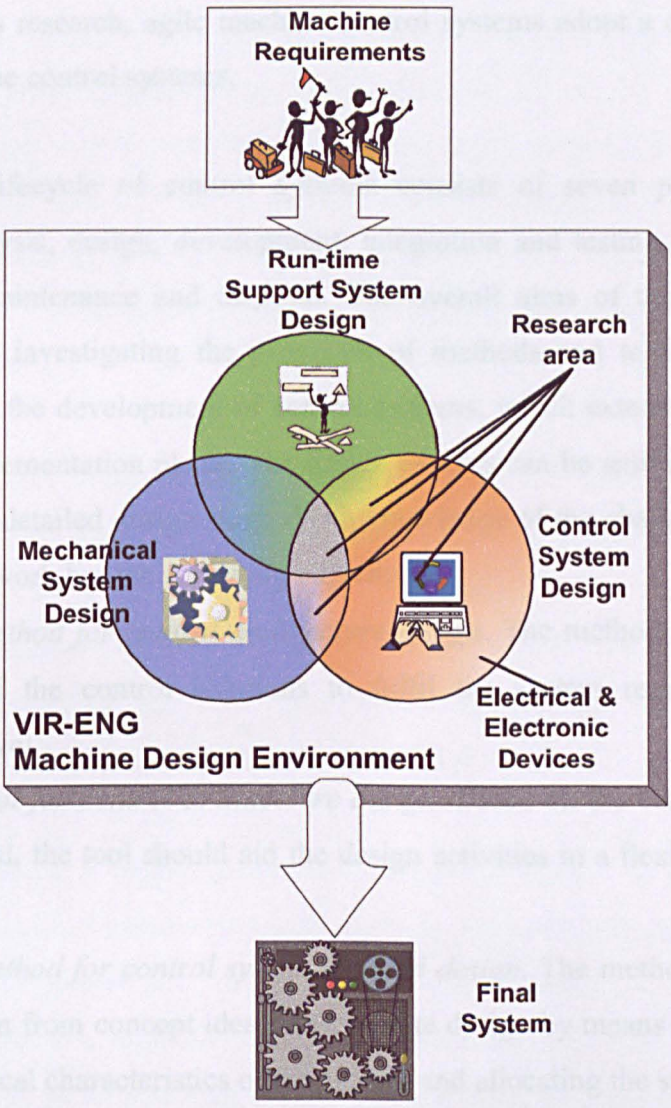

Figure 1.1 Research Areas

1.3 Research Aim and Objectives

The aim of this research work is to investigate methods and tools for the realisation of agile manufacturing machine control systems. In order to achieve this goal, it is necessary to adjust the way that control systems are composed, integrated, reconfigured, and reused. Recent advances in software technology have demonstrated

the potential to revolutionise control system realisation. In particular, new component-based architectures encourage flexible "plug-and-play" reconfigurable systems. Inspired by the same concept, new agile control systems can be built from prefabricated components, including software components and hardware components. Therefore, in this research, agile machine control systems adopt a component-based approach to realise control systems.

Generally, the lifecycle of control systems consists of seven phases: planning, requirement analysis, design, development, integration and testing, implementation, operation and maintenance and disposal. The overall aims of this research study mainly focus on investigating the provision of methods and tools to support the phases related to the development of control systems, which extend from the design phase to the implementation phase. The design process can be split into a conceptual design stage and detailed design stage. With knowledge of the challenges outlined in Section 1.1, this work has the following objectives:

1. *Propose a method for control architecture design.* The method should guide the generation of the control solutions to fulfil the system requirements at the conceptual level.
2. *Produce a tool for control architecture design.* Based on the control architecture design method, the tool should aid the design activities in a flexible and efficient manner.
3. *Propose a method for control system detailed design.* The method should aid the transformation from concept idea into concrete design by means of specifying the internal physical characteristics of the system and allocating the system resources.
4. *Develop techniques and tools for control system implementation.* This involves translating the detailed design into hardware, communications and executable software.

This work also seeks to achieve a set of secondary objectives including:

5. *Integration of the design process with other design environments.* The control system design process is not a stand-alone process but needs to be considered as a related entity with other design activities such as mechanical system design to ensure the completeness or integrity of the entire machine system.
6. *Produce a proof of concept demonstration system to demonstrate and evaluate the practicability of the approaches and the associated tools.*

1.4 Thesis Organisation

The thesis is organised into eight chapters. An overview of the thesis follows.

Chapter 2 outlines some key elements of the current manufacturing development and the associated control system design methodologies. A survey of contemporary control architecture design and description languages adopted in control system design is presented. Based on this review, a suitable description language has been identified. The existing software infrastructures are also discussed. Programming languages that can be used to implement control systems are reviewed from a control engineering perspective.

Chapter 3 reviews the underlying design philosophies behind VIR-ENG. A reference model for the agile machine system is introduced. As a part of agile machine development, the development process is described from a control system development perspective. Through analysis of the whole machine development process, the development process for the control system has been formulated. The key elements of control system development are identified.

Chapter 4 introduces a methodology for the control system architecture design. Inspired by information technology design methods, the new methodology contributes some key features specifically for the control system design. UML as a widely accepted modelling language in software engineering has been adopted as the modelling language in architecture design. Additionally, an associated support tool named CADE is described.

Chapter 5 addresses detailed design and the development of the underlying mechanism to implement the detailed design, based upon the component-based strategy. Detailed design transforms the architecture design into the specific hardware and software structure. The detailed design implementation is facilitated by the underlying mechanism, whose model is described and implementation issues are addressed. Two supporting tools are introduced to aid the development process.

Subsequently, these elements form a complete environment to support the implementation phase.

Chapter 6 covers integration within the entire VIR-ENG development environment, which forms a complete environment to support the agile machine design process. It includes integration with the mechanical design environment and the runtime support environment.

Chapter 7 covers the process to employ the proposed methods and tools on a demonstrator cell. The demonstrator of the VIR-ENG project was built within Euromation (a Volvo Group company). Other collaborators with the Framework IV ESPRIT sponsored project included Delfoi, Honeywell International, University of Skovde and Gothia Science Park. This demonstrator was conceived and implemented to represent real production facilities. The main objective of the demonstrator cell was to provide a comprehensive view of the facilities and capabilities of the VIR-ENG approach and tool-set.

Chapter 8 summarises the work undertaken, the main conclusions and contributions to knowledge and the advances resulting from this research.

Chapter 2 Literature Review

2.1 Introduction

Manufacturing industry might be on the verge of a major paradigm shift. This shift is likely to take us away from mass production, beyond lean manufacturing and into a world of Agile Manufacturing. The need for the machine system, as a vital element of manufacturing systems, has led to the concept of the agile machine systems (Kusiak and He, 1997). It is arguable that as one of the most influential elements in the machine system, the control system needs to demonstrate the most agility; this could be referred to as the 'agile control system' (Young *et al.*, 2001). A major difficulty experienced in the construction and evolution of agile control systems is the lack of systematic approaches to enable change capability and incorporate multidisciplinary techniques development (inside and outside the control domain) in a cost-efficient and timely manner. These were the goals of this research. In this regard, this chapter reviews:

- the development trends in machine systems and current enabling technologies;
- concept design methods and means to fulfil requirements;
- available software architectures and languages in this research area to implement the concept design and take advantage of enabling technologies.

2.2 Trends in Machine Control

Prior to the 1950s, machine control systems were characterised by the use of mechanical mechanisms (such as mechanical linkages, gears and cams). These mechanisms were best suited to automating operations in large scale batch manufacturing, such as in transfer lines designed to produce automobiles in large quantities and with only minor operational change necessary due to limited product variants (Koren 1983).

The introduction of numerical control (NC) in the 1950s signalled a new era in automation (Koren 1983). Since that time, various types of manufacturing machine requiring fixed sequences of operations deployed control technology in the form of electrical relays, discrete pneumatic logic devices and hydraulic hardware elements. Towards the end of that decade, semiconductor devices began to dominate as the preferred choice of technology for building the logical elements of control systems.

During the 1960s and 1970s major advances in semiconductor technology (leading to the availability of microprocessors and other LSI) led to the development of CNC (computer numerical control) systems for machine tools, industrial robots and PLCs (programmable logical controllers). Such advances have been progressively used in manufacturing industry (Pritschow 1990, Olsson *et al.* 1993) to provide machines with great operational flexibility, which in turn has promoted opportunities to automate small batch manufacturing process, in order to cope with the growing competitiveness of the global market (Waller 1983). In the late 1970s, the availability of such flexible machines led to the development of flexible manufacturing systems (FMSs) to automate the production of families of products (Inamoto 1990). Consequently, the system can produce various combinations and schedules of parts or products instead of requiring production in batches. This is a case of soft variety, so that the amount of changeover required between styles is minimal.

In the late 1970s and the 1980s, the initial visionary work on the application of computers to manufacturing was done by a handful of people including Bjorke (1979), and Merchant (1980). They created the concept of computer integrated manufacturing (CIM) as the way to automate, optimise, and integrate the operations of the total manufacturing system, which includes robotics and unattended flexible manufacturing systems. Its primary object was to reduce factory floor labour costs. Since the mid-1980s, the focus has been the pursuit of greater flexibility, elimination of excess inventory, shortened lead-times, and advanced levels of quality in both products and customer service. Such practice has been classified as 'lean manufacturing' (Sheridan 1993). As the cost of the technology progressively dropped and its industrial package and usability improved, it became practical to place more intelligence close to process sensors and actuators on individual machines. Such decentralised control, based on low cost microprocessor technology, has penetrated

virtually every industrial sector (Northcott 1988). In the drive towards the implementation of CIM, the integration of a diverse range of machine control systems as well as the integration of entire manufacturing systems has been increased emphasis, which today embodies the notion of achieving the coherent interworking of people and flexible machines, by supporting them with interoperating computer systems (Weston *et al.* 1994).

The typical integration solution was to directly interconnect digital I/O between the various device controllers (Allmendinger 1987). In this type of set-up a PLC was often used to provide primary sequences of more complex custom devices (such as motion controllers, robots or machine tools) using its configurable I/O system. In order to improve on such solutions many system builders advocated the integration of PCs, dedicated machine controllers and PLCs in a triangular hierarchy (Kendrick 1987, Yingst 1987, Alvers 1988). The PLCs and machine controllers are utilised for time critical control while the PCs perform monitoring, data collection and data management functions. Later, this solution has been further improved by means of using a control network to replace the I/O system (Manji 1992). To address the issue of integration in a more ordered way, numerous vendors, users, and university research groups have discussed and worked on various cell control concepts (Paidy *et al.* 1991, Doiteaux *et al.* 1991). Practical cell control systems typically involved the use of industrial microcomputers to (i) connect to a high level network; (ii) provide a cell-level man-machine interface; (iii) provide reprogrammable integration of the cell's individual device controller, to provide maximum operational flexibility.

In 1991, concerns about competitiveness in a new global manufacturing environment led US industry and the Federal government to begin a joint study. As a result, the Agile Manufacturing Enterprise Forum (AMEF) was formed and the concept of agile manufacturing was introduced (Sheridan 1993, Struebing 1995, Goldman 1994). For many, 'Lean manufacturing' and 'Agile manufacturing' sound similar, but they are different. Lean manufacturing is a response to competitive pressures with limited resources. Agile manufacturing, on the other hand, is a response to complexity brought about by constant change. Some researchers contrast flexible manufacturing system (FMS) and agile manufacturing system (AMS) according to the type of adaptation: FMS is reactive adaptation, while AMS is proactive adaptation. Quinn *et*

al. (Quinn *et al.* 1997, Wyatt *et al.* 2000) successfully demonstrate the proactive adaptation in an agile manufacturing work-cell.

In order to deliver agility for manufacturing machinery, the Agile Reconfigurable Manufacturing Machinery Systems (ARMMS) programme has identified several strategies (Moore *et al.* 2002, De Vin *et al.* 2002). One key characteristic of such machine systems is reconfigurability. Because of the last few years' development, some technologies that are necessary enablers for reconfiguration have emerged. In machine control, these are modular, open-architecture controls that aim at allowing reconfiguration of the controller (Koren *et al.* 1998). In machine mechanics, they are modular machine tools that aim at offering the customer more machine options (Mehrabi *et al.* 1997). These emerging technologies show a trend toward the design of systems with reconfigurable control systems and reconfigurable mechanical systems. These are necessary but not sufficient conditions for a true agile machine system. The core requirement to deliver an agile machine system is an approach to reconfiguration based on systematic design combined with the simultaneous design of open-architecture reconfigurable controllers and reconfigurable modular machines. In addition, as a part of a manufacturing system, control systems should adopt a systematic way of integrating control systems as well as integrating with other manufacturing system components, which is important for maintenance and reconfiguration in the future (Weston 1989). Naturally such advances in practice, when designing and building machine systems, will need to build upon and deploy recent advances in technology; such as availability of improved computer aided design tools and the ability to distribute and embed computing capability into typical component building blocks of automated manufacturing systems (Harrison 1995). The 'VIR-ENG' project especially addressed the development of highly integrated design, simulation and distributed control environments for building agile modular manufacturing machine systems which offer the inherent capacity to allow rapid response to product model changes and feature variants (Pu and Moore, 1998).

2.3 Enabling Technologies for Agility of Machine Control Systems

Agile manufacturing has attracted attention from both the academic and industrial communities. Extensive programmes are being conducted on relevant issues to propagate agile manufacturing concepts, to build agile enterprise prototypes, and eventually to realise an agile industry. The AMEF has sponsored several major conferences and has created at least 18 ongoing 'focus groups' to explore further various aspects of agility and the infrastructure needed to support them. Certain enabling technologies, for example the standard for the exchange of product model data (STEP), concurrent engineering, and organisational and behavioural changes are Critical to successfully accomplishing agile manufacturing (Cho 1996). Most researchers have focused on the management and organisational aspects of agile manufacturing. However, in order that agile manufacturing can have an appreciable effect, the associated manufacturing system must be up to the challenge (Koepfer 1995). Within the scope of this research, the following sections are primarily concerned with the enabling technologies for machine control systems in the context of agile manufacturing. Four categories are broadly identified, namely, information infrastructure, control networks, open architecture control systems and component-based approach.

2.3.1 Information Infrastructure

The initial development of information infrastructure was driven by Computer Integrated Manufacturing (CIM), which was the phrase used to describe the complete automation of a manufacturing plant, including automation of manufacturing system in the factory and computerisation of the manufacturing support system (Mikell 2001). As the integration emphasised, it required communication over all factory levels from business operation down to the control of real-time operations in a flexible manner (Merchant 1985). Its development involved standardising protocols and associated service functions.

When the General Motors Corporation began its Manufacturing Automation Protocol (MAP) effort in 1980, they used the EIA-1393A draft standard proposal as the basis

for a more generic messaging protocol that can be used for NCs, PLCs, robots and other intelligent devices commonly used in manufacturing environments. The result was the Manufacturing Message Format Standard (MMFS). MMFS was used in the MAP Version 2 specifications published in 1984. With the objective of developing a generic and non-industry specific messaging system for communications between intelligent manufacturing devices, a standard called the Manufacturing Message Specification (MMS) has been proposed based upon the Open Systems Interconnection (OSI) networking model (ISO 1990). The MAP/MMS solution was a heavy one and, if conceivable for automotive large factories, was hardly adaptable to small production units for reasons of price, complexity, and rigidity of legacy control systems.

Since August 1996 OPC version 1.0 was released, OPC becomes a de-facto open standard for sharing manufacturing information in an enterprise-wide manner (Chisholm, 1998; Harrison, 1998). It is based on the Microsoft technologies of OLE (Object Linking and Embedding), COM (Component Object Model) and DCOM (Distributed Component Object Model). It provides 'plug-n-play' connectivity and interoperability between disparate automation devices, systems and software, from the shop floor to enterprise-wide systems.

Recently, an internationally approved standard of interest to information integration discussions is ISA-S95. S95 was being developed as a multi-part standard that defines the interfaces between business and manufacturing systems. Although S95 Parts 1 and 2 (ISA-S95 2000, ISA-S95 2001) did not define a formal protocol or detailed format for information exchange between systems, they did provide a base on which exchanges can take place. The S95 committee has already begun working on Part 3 to define models for the disparate collection of activities that must occur between manufacturing operations and business logistic systems to finally achieve the integrated enterprise vision. XML (eXtensible Mark-up Language) schema definitions have not been finalized by Web standards organizations, but that has not held back several S95 committee member companies, who are 'betting' that XML will be the de-facto standard for exchanging information. These companies have embarked on projects to create solutions to support business and manufacturing information sharing using XML schemas. In fact, XML is such a strong contender to be the enterprise-

wide connectivity method, which subsequently, the OPC Foundation expects to include XML to be a part of new OPC specification (OPC, 2001).

Another under-developed standard is to specify CORBA for machine control. This was undertaken by the Manufacturing Domain Task Force (DTF) within the Object Management Group (OMG) (OMG 1998). By taking advantage of CORBA technology, it intended to achieve interoperability within the manufacturing environment and a distributed object environment within the machine control environment.

Currently, OPC seems to be the only available mature technology for agile control systems. By extending the existing OPC DA (Data Access), OPC can be adopted to fulfil the requirements of Data Exchange, which provides interoperable data exchange communications across Ethernet networks between HMIs, controllers and other intelligent devices.

The benefits from the development of information infrastructure can be identified as (Duffie 1996):

1) Openness – relying on well structured and widely implemented interface protocols, so that anyone can use and offer services through an agile infrastructure.
2) Availability – the capability to access service from control unit as well as around the world using the same protocols.
3) Extendibility and graceful degradation – services can be added, removed or substituted at any time, with incremental changes in performance.
4) Compatibility – with legacy systems through encapsulation.

2.3.2 Control Networks

The control networks represent the core technology of the control system communication, which link the various control elements. They include not only fieldbus networks associated with low-level device communication, but also Internet/Intranet and Ethernet associated with high-level information flow. Recently,

Ethernet has penetrated into the control level (Perry 1999). The adoption of an international standard, which defines the physical connection of devices, communication protocols, messaging mechanism and application profiles, was advocated by the international field-bus initiative (McKenna 1992). This developing strategy relies on entirely embracing standards that are emerging and being accepted commercially. Although distributed intelligence is one of the by-products of fieldbus systems, it can contribute to the agility of the control system.

The primary considerations of fieldbus systems and control systems, in terms of the physical data transmission, are the required response time, the amount of data being transferred, the frequency of transmission and the behaviour under fault conditions, which requires rigorous specification, design and verification processes (Rodd *et al.* 1998). In this regard, Seung (2000) established an experimental model of a Profibus-based manufacturing automation system in order to examine the performance characteristics of the industrial communications network. Using the experimental model, this study evaluated the latency characteristics of a message transmitted through the fieldbus message specification services. The message latency was measured at each sublayer of the Profibus protocol stack. Broadly, it provided a measured method to evaluate the real-time capability of fieldbus systems. Tovar (Tovar and Vasques 2001) used another fieldbus system, named WorldFIP, to implement distributed computer-controlled systems. He described how WorldFIP handles two types of network traffic distinguishing periodic and aperiodic traffic. Most importantly, he provided a comprehensive analysis on how to guarantee the timing requirements of the real-time traffic.

The major benefit of fieldbus systems is to allow distributed control solutions to be realised cost-effectively and efficiently. The system is truly interoperable between different manufacturers' devices, as far as it conforms to the same standard specification (Zielinkski 1999). When adopting such systems, not only the cost of wiring and installation is greatly reduced, but also the maintenance of plant and equipment is simplified. For control systems, the control elements can be considered truly distributed not only in conceptual terms but also in physical forms. The distributed "intelligence" of the control system enables the field components to dynamically collaborate to satisfy both local and global objectives. Since such

decentralised machine control systems have a virtual organisational structure, its elements can be more flexible and rapidly re-organised, which in turn reflects the agility of the control system. "Intelligence" of control systems can now be distributed to the device level, which provided a new dimension in building manufacturing machines and production systems (Moore *et al.* 1993; Moore *et al.* 1994b; Warwick 1994). With fieldbus, many of the functions contained in traditional control systems such as function blocks, deterministic scheduling, a distributed database, trending, and alarming can migrate to the devices and the network itself (Glanzer 1998). This distribution paradigm confers more capacity to the field components, which is needed in an agile manufacturing context (Weston 1998). Wang (Wang *et al.* 1998) implemented a proof–of-concept prototype based on fieldbus, which showed promise for improving the agility of the manufacturing system. K. W. Young (Young 2001) realised the agile control system by applying three types of fieldbus system in the SALVO demonstration.

However, the fieldbus system cannot achieve the requirements of agile manufacture alone, it needs to be integrated with information systems, which is dominated by Internet/Intranet and Ethernet. Because skilled and specialised personnel are typically not available at site, it is necessary to remotely access the intelligent devices via the Internet or Intranet (Fantoni and Chatelet 2000). Internet and Intranet solutions are increasingly used in the industrial automation scenario at the factory and cell level, and a new generation of applications for manufacturing and process control environments are appearing on the market based on popular Internet tools and protocols. Currently, one fieldbus system cannot be used alone in an optimised control system architecture. In fact, different and incompatible demands are placed on bus systems at different levels of automation control (Perry 2001). Therefore, it is necessary to use fieldbus technologies intelligently, and to mix and match technologies to provide a cost-effective solution to complex control problems.

The benefits from the development of control networks can be identified as:

1. Openness – relying on well standardised fieldbus hardware and communication protocols, so that control solutions from different vendors can be integrated.
2. Distribution – not only logically, but also physically distributing control functions.

3. Extendibility – control units can be physically added, removed or substituted at any time.

2.3.3 Open Architecture Control Systems

One of the major impediments to agile manufacture for many industries is the lack of an open environment for rapid integration of control functions. Some researchers have pursued an "open" machine control system to facilitate the creation of 'agility' in control systems. To classify control systems, OMAC (Bailo and Yen 1997) defined the degrees of openness within a control system, shown in figure 2.1. On the far left would be "black box" controllers, most likely designed by an OEM or integrator for a specific application. The first level of openness beyond the proprietary control is the open environment controller. According to the definition, the open environment in today' s market is the IBM Personal Computer compatible hardware platform with Microsoft Windows operating system. However, it requires extensive engineering effort to integrate these hardware and software components into a functional control system. Even though modularity and scalability are not there for the open environment controllers, users can certainly gain benefits from the open environment because of the choice of available components from multiple vendors and greater freedom of configuring the system to meet particular application needs. A step forward to openness is to define a common set of APIs. With the availability of common APIs and products conforming to the APIs, it is possible for users to re-configure control systems without extensive engineering effort. Then "plug and play" and "scalability" become a reality. The final level of openness removes proprietary hardware elements from the control system. The "Open, Modular Architecture Control" level can be considered as a software based controller with generic processors running software control modules without special hardware, such as motion control cards and discrete logic control cards that are plugged into the controller backplane.

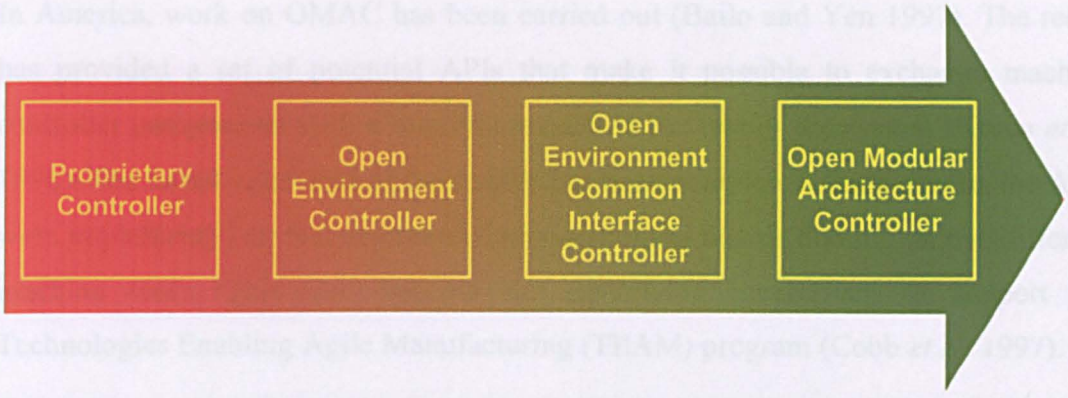

Figure 2.1 Controller Openness Continuum

In Europe, the ESPRIT III project OSACA (open system architecture for controls within automation systems) has worked out the necessary specifications and jointly developed the software modules for the system platform (Lutz and Sperling 1997). Within this platform, the standardised communication system enabled uniform information interchange between all the control specific modules (Lutz and Sperling 1995). The software package contained a universal set of components for operating systems on PC hardware under MS-Windows and an information interface for connecting different control systems (Szabo and Proctor 1997). To prove the concept, the architecture has been implemented in a NC demonstration cell.

UMC (Universal Machine Controller) (Weston *et al.* 1989b; Weston *et al.* 1989c) was developed by the Modular System Group (MSG) of the Manufacturing System Integration (MSI) Research Institute at Loughborough University. Initially, it was intended to develop methodologies to advance the use of distributed modular machines (Harrison *et al.* 1987). In the end, it provided an open control approach and a set of run-time software modules, which could be integrated with the control system, and a set of facilities, which could be used to assist in the configuration and management of the runtime system (Harrison 1991). The concepts and tools have been demonstrated in packaging machinery (Goh and Moore 1994), a gantry crane (Moore *et al.* 1994a), an agricultural vehicles system (Anon 1996), and a beverage cans manufacturing application (Anon 1997; Anon 1998).

In America, work on OMAC has been carried out (Bailo and Yen 1997). The result has provided a set of potential APIs that make it possible to exchange machine controller components with a minimal impact on the rest of the system (Koren *et al.* 1998). The initial validation of these APIs has been completed. Furthermore, the APIs were implemented on two commercial motion control boards that run on two different machine tools. This also became the underlying mechanism to support the Technologies Enabling Agile Manufacturing (TEAM) program (Cobb *et al.* 1997).

In addition to exhibiting openness, the controllers of future agile machines should be distributed. The above-mentioned projects have not explicitly considered these issues, although they have allowed control of distributed processes to some extent. In this respect, it is worth stressing the EU-sponsored project HEDRA (Heterogeneous and Distributed Real-time Architecture) (Thielemans *et al.*, 1998), which deliberately addressed a heterogeneous distributed real-time architecture for robot and machine control based on an existing system called VIRTUOSO.

The benefits from the development of open control systems can be identified as:

1. Openness – relying on a computing platform that is readily available to all control builders, has available development software, supports high speed processing and readily interfaces to a vast array of other devices.
2. Upward migration path – providing an upward migration path in both hardware and software, as computer technology advances in order to keep the machine updated.
3. Compatibility – ability to run third party, off-the-shelf software without comprising the control system.

2.3.4 Component-based Approach

The component-based development strategy has been promoted by 'component-based technology', which has emerged and has begun to be used in the commercial marketplace (Wallnau, K. 1998). For many years, machine builders have pursued the migration from custom development to assembly from pre-fabricated components (Harry1997). The major benefit from component-based development is seen as an

enabling element for the delivery of "agility", which then allows machine builders to produce custom machine systems in a cost-efficiently manner so that systems can be easily reconfigured. A reconfigurable component-based system promises a step forward to meet changing requirements (Weston 1999).

Through enterprise wide decomposition, the whole system can be divided into recognisable, manageable and consistent segments, which can be separately implemented. If the segments conform to explicit models, these segment solutions can be reused and considered as system components. From the enterprise wide perspective, the component denotes a reusable and interoperable part of an enterprise system. It can be referred to as a system block (Kosanke and Vernadat 1996), reusable object (Brown 1996), system holon (VanBrussel *et al.* 1996) and so on.

Reconfigurable mechanical elements of modular machine systems, which include linear and rotary mechanical modules, conveyor systems, transfer units, etc., have been developed and used for several decades. Recent developments of fieldbus technology, smart sensing technology, and modular software systems facilitate the development of reconfigurable control system elements. These enabled machine building for realising the overall machine system based on 'components', which could, be for example sensor components, control components and mechanical modules, and so on. Intelligence can be embedded into 'components', where this is appropriate, to realise attributes of autonomy and co-operativeness (Deen 1993). Components can be composed or aggregated together to form components that are more complex. Adopting the component-based paradigm, the specified machine system can be constructed based on these components (Pu and Moore 1995a).

The basic concepts of component-based machine system development have been greatly influenced by the trends in component-based software engineering (Brown and Wallnau 1996). Methods covering analysis, design, and implementation of component-based software system have emerged. The fundamental idea behind the use of software components is based on the 'buy, don't build' philosophy advocated by Fred Brooks (Brooks 1987). Whilst the design of software components was founded largely on the notion that certain parts of software elements reappear regularly, these generic software elements can in many cases be written once, but

reused repeatedly. Modern software technology has enabled software component to be reused in the executable form rather than by copying and pasting the source code. In particular, ControlShell (1998) was known as a next-generation object-oriented software framework for real-time system development. It includes graphical Computer-Aided Software Environment (CASE) tools, an object-oriented component-based architecture, and integrated data management, which were developed jointly at Real-Time innovations, Inc. and Stanford University. Based on the component-based approach, ControlShell applications are built from objects called Composite Object Groups (COGs). It implements complicated control systems by means of progressively aggregating basic control units or objects.

Although machine control systems consist of software elements and the control system increasingly becomes software intensive, control software is not equal to common software. As mentioned before, the critical time constraint is an important attribute that distinguishes control system components from common software components (Andy and Pete 1997). However, software components are not the only element in the entire control system; hardware components are also necessaryand important elements of the control system. Naylor and Volz (Naylor and Volz 1987) introduced the software/hardware component as integral entity of manufacturing control. A software/hardware component encapsulates the physical devices and the software component responds to low-level control of the physical devices. Chaar extended the work of Naylor and Volz by allowing hierarchically constructed and assembled software/hardware components and presented several implementations of software/hardware components (Chaar, 1990; Hadj-Alouane, 1990).

In identifying appropriate paradigms for machine design and control, Pu J. *et al.* introduced some preliminary thoughts, namely a component/image based approach, which in fact served as the conceptual framework for the methodology developed in the European project "VIR-ENG" (Pu and Moore 1995b). 'Components' could be modular physical machine units, software components, or electrical hardware components. This idea encouraged component reuse (including software) and application knowledge reuse, and facilitated the reconfigurability of the machine or machines system in a timely and proven fashion.

Overall, the benefits from the component-based approach are the use of component-based systems engineering concepts. These concepts have shaped the design of a component-based approach to the life-cycle engineering of machine systems. It can lead to the seeking of a flexible integration structure to sets of interoperating machine components, where this structure positively enables change rather than inhibits it. Since a component-based approach can be adopted by control system development, as well as mechanical system development, it aligns the re-configuration strategies of control systems with mechanical systems. In addition, for control systems, this approach could reduce cost, risk and lead-time by means of building control systems from pre-built, proven components.

2.4 Conceptual Design

It has been recognised that the first step in system design is to create a conceptual design of the 'system to-be' (Cooke 2001). The software engineering community, and in particular the automatic system community has been concerned with this research. Mainly, conceptual design involves choice of a suitable control architecture and the use of an architecture description language associated with a design methodology for architectural development (Chen 2000). Control architecture refers to the architecture of a specific control system. It is the result of a design process for a specific control system and specifies the functions of control components, their interfaces, their interactions and constraints. This specification is the basis for detailed design and implementation. In order to successfully achieve these goals, control architecture design relies on the adoption of a description language to represent and document design artefact associated with a clear and systematic methodology. Architectural development is considered as a "good way" to achieve high levels of system quality, especially those related to the non-functional requirements. It offers promising perspectives, which are necessary for control system development. Some of the advantages include rapidly building the system, predicting global qualities of the final system, enhancing modularity and less faults by restricting design variability (Clements 1996).

2.4.1 Control Architecture

There is no standard, universally accepted definition of the term 'control architecture'; it is a field in its infancy, although its roots run deep into control engineering. The definitions of architecture in the Oxford dictionary are:

1. the art or science of designing and constructing buildings,
2. the style of a building as regards design and construction,
3. buildings or other structures collectively,
4. the complex structure of something.

According to this definition, architecture mainly refers to the building domain, for example the style of a building, and it also refers to the knowledge and styles, for example the conceptual structure and logical organisation of a computer or computer-based system. For the control system, the objectives of architecture can be considered as:

- An abstraction to describe complex dynamic systems by providing simple models. The architecture helps the designer to define and control the interfaces and the integration of the components (Zachman 1987).
- During the design or re-design process, the architecture is a means of communication. It may provide several abstract views on the system, which serve as a basis to clarify each party's perception of the problem area (Philippe 1995). The architecture also helps to reduce the impact of changes to as few modules as possible. The architectural model allows the control system to focus on the areas requiring major change.
- When the system is adapted to new uses, the architecture indicates the most vital system components and constructs that should not be violated. Violating the architecture - similar to removing load-bearing walls in a house - decreases the system's ability to change gracefully with changing constraints and requirements (Perry and Wolf 1992).

Although there is no universally accepted categorisation, four basic control architectures have been identified as centralised, hierarchical, heterarchical and hybrid (Dilts *et al.* 1991).

2.4.1.1 Centralised Control Architecture

Centralised control architectures are typically a single controller running on a single computer directly controlling everything. It includes a centralised data repository with a single data access protocol. Johnson (Johnson and Baker 1990) described a centralised control system named CIMPICITY, which is a GE Fanuc product. Maimon (Maimon 1987) presented a centralised controller for a flexible manufacturing system that is internally decomposed.

The centralised control architecture advantages (Hammer 1987) are:

1. Less communication among controllers, the main communication is between controllers and device drivers,
2. Data easily collected,
3. Less problems for global optimisation,
4. Easy verification. Since only one program, one status interface, one control interface needs to be evaluated.

The disadvantages of the centralised control architecture are:

1. Vulnerability to failure. If one little thing goes wrong, it will cause the whole system to fail.
2. Sudden performance degradation. Generally, if one part does not work, the whole system does not work at all.
3. Hard to extend. When the system grows, the computing resource loading is always required in the host computer. If the demand is too large, there may be no way to extend it.
4. Hard to maintain. Since every thing is loaded in the central controller, it can become too complex to debug and maintain.

2.4.1.2 Hierarchical Control Architecture

One important feature of hierarchical control architecture is that the control function is decomposed into several modules arranged in a pyramid. It is designed so that there is a hierarchy and a master/slave relationship between higher and lower levels of

control. In the strict hierarchical control architecture, the exchange of commands is not allowed between the same level controllers. Normally, sensory data flow is in an upward direction from units at the lowest level to higher-level supervisory controllers and command data are generated and sent in a downward direction from supervisory controllers to units. Since most systems are physically organised in a hierarchical structure, it is easier for developers and practitioners to adopt a hierarchical control architecture.

Albus (Albus *et al.* 1981) described a hierarchical robot control system and identified three basic guidelines for developing manufacturing control hierarchies: (1) Levels are introduced to reduce complexity and limit responsibility and authority, (2) each level has a distinct planning horizon which decreases as it goes down the hierarchy and (3) control resides at the lowest possible level. The robot control system introduced the concept of integrating hierarchically decomposed commands from higher levels with status feedback from lower levels, to generate real-time control actions.

The National Institute of Standards and Technology (NIST) has been developing an Automated Manufacturing Research Facility (AMRF) since 1981. As a part of their research, a five-level hierarchical control architecture has been developed at NIST (Johnson *et al.* 1990; Jones *et al.* 1985; Jones *et al.* 1986). A new project at NIST called the Manufacturing Systems Integration (MSI) project has been established to refine and extend the original NIST architecture work (Jones *et al.* 1990; Jones *et al.* 1992; Joshi, *et al.* 1990). One of the main contentions of the MSI group is that the five level hierarchy imposed by the original NIST architecture is too rigid in structure. In response, the new MSI architecture only defined three types of controllers: equipment, workcell, and shop. A hierarchical structure of arbitrary depth can be constructed using these three types of controllers. However, the primary focus of the MSI projects has been to develop standards for system (i.e. controller) interconnection, so that the whole model can help to facilitate enterprise wide integration (Jones *et al.* 2000).

Chryssolouris *et al.* (Chryssolouris *et al.* 1988; Chryssolouris *et al.* 1991) presented a work centre level controller, named MADEMA (Manufacturing Decision Making),

which assumed four levels of hierarchy: factory, job shop, work centre, and resource. MADEMA obtains work requirements (type, quantity, release times, and due dates of jobs) from the job shop level, determines feasible alternatives of task-resource pairs, relevant criteria and the consequences of alternatives with multiple criteria, then applies decision-making rules and then selects the best alternative.

An architecture (Dornier 1990; Dornier 1991) was proposed for the European space automation and robotics control system. Associated with the architecture, a formalised methodology named structured analysis and design technique (SADT) was provided for architecture development. Cho and Wysk (Cho and Wysk 1993; Cho *et al.* 1997) developed a three level hierarchical shop floor control system. The levels in the system correspond to shop, workstation, and equipment levels in the hierarchy. In common, the developments focused on middle level workstation controllers. Additionally, the AND/OR graph was used for representing alternative process plans.

The advantages of hierarchical control architecture can be summarised as:

1. Natural modularity. Each controller is considered as a discrete module, facilitating incremental development, making the system easier to understand and maintain, and allowing controller development using templates.
2. Easy to extend. The system can be extended by adding controllers and computers in the hierarchy as needed.
3. Graceful degradation. If some part of the system fails during system operation, in a well designed hierarchical system, only one part of the system needs to be stopped
4. Allowance for different operational frequencies of controllers on different levels of the hierarchy. Typically, a hierarchical control system has different operational frequency across the entire system. Controllers closer to hardware at a low-level run higher frequencies.

Some disadvantages of hierarchical systems, include:

1. Increased communications links are needed between controllers. This is not necessary for centralised controllers.
2. Difficult to integrate system-wide service functions.

3. Difficult to test. Errors may occur in the interactions between controllers, which makes for testing problems. The fact that the controllers are distributed among processes or computers, conventional debugging tools, which are effective with centralised control, are not so helpful in finding such errors in such systems.

2.4.1.3 Heterarchical Control Architecture

In the pure heterarchical control architecture, each controller has no superior and no subordinates. The main object is to remove the rigid master/slaves relationships associated with hierarchical architecture. This is being driven, to a significant extent, by the growth in the density and level of distribution of computing resources, the demands for exchanging data and control information between physically separated entities and the need to accommodate complex interrelationships between entities without superimposing the additional complexity of a rigid control hierarchy. Controllers interact by issuing requests for bids, making bids, and entering into contracts to do work. Dilts *et al.* (1991) discussed heterarchy in the context of a comparison of types of architectures and Hatvany (Hatvany 1985), and Ting (1990) adopted this architecture. There are also several other proposed heterarchical control architectures (Rana 1988). Recently, a paper presented a structured methodology for design and implementation of heterarchical shop floor control systems using an industrial framework, Windows-DNA (Windows-Distributed Internet Applications) (Ozgur *et al.* 2000).

The common feature of heterarchical architecture is to minimise or totally eliminate the global information. The behaviour of such systems, comprised of highly autonomous entities, can seem chaotic (Duffie and Prabhu 1994; Hogg and Huberman 1991). In manufacturing system control, this can be a by-product of the application of heuristic-based intelligence. Several papers (Duffie 1987; Duffie *et al.* 1988) present a part-oriented heterarchical control system in which each individual part and machine is represented by a system entity. Part entities have knowledge about the processing that they require and machine entities have knowledge about the processing that they can perform. Part entities "broadcast" processing requirements over the system network and machine entities similarly broadcast processing availability. When a

match is found, the part and machine entities negotiate and, once an agreement is made, the part is transported to the machine and processing begins. There is a perception that the performance of heterarchical system cannot be characterised and cannot be guaranteed because such systems lack master-slave relationships and the ability to force order and cooperation through a hierarchy of control. Therefore, in spite of their potential benefits, industry has been reluctant to adopt heterarchical architectures. In order to increase determination, a new local control method based on closed-loop feedback rather than heuristics has been developed for a class of heterarchical systems in which part entities must cooperate through communication to determine their individual arrival times at a machine (Prabhu and Duffie 1995).

Upton (Upton *et al.* 1991) presented some issues which have arisen during the course of a research project which examined methods for operating large computer-controlled manufacturing systems, with over 50 or so CNC machines. Based on a simulated manufacturing system with a standard part flow and multiple possible machines (each with a different processing time for similar parts), it was recognised that the distributed architecture dispatching jobs as a centralized controller might use the best machines when idle, and progressively less effective machines when busier. It has been pointed out that further research in process planning, communications, and on-board information processing is required in order to make the heterarchical architecture feasible for shop-floor control.

Advantages of the heterarchical control architecture can include:

1. Strong modularity. Each controller can be treated as a software module.
2. Readily extended. The system can be extended by adding controllers and computers (new modules).
3. Graceful degradation. If some error occurs in one controller during system operation, only the controller that has the problems needs to be terminated.
4. Full local autonomy.

Disadvantages include:

1. Requiring a higher network capacity. This is an essential requirement to cope with all the soliciting, bidding, and contracting.

2. Unpredictable system behaviour. Predicting what the heterarchical architecture will do (for example, which controller will do which tasks and when a task will be done) is typically difficult. It is often not clear if the solicit-bid-contract procedure will reach a conclusion.
3. Hard to optimise global behaviour, however performance should still outstrip distributed.

2.4.1.4 Hybrid Control Architecture

A investigation in control architecture area (Rogers and Brennan, 1997; Brennan and Norrie, 1998) indicated that neither the hierarchical nor the heterarchical of the control architecture spectrum provided the most appropriate control architecture solution, but that hybrids of hierarchical and heterarchical control architecture show the most promise. (Sometimes this is refereed to as modified hierarchical control architecture or semi-heterarchical control architecture.)

The partial dynamic control hierarchy (PDCH) was proposed as a control architecture, which combined the features of both hierarchical and heterarchical architectures (Brennan *et al.* 1997). Later, it was further extended by means of adopting PDCH and IEC-1499 function blocks to implement a real-time distributed manufacturing environment (Wang *et al.* 1998). At the low-level, the individual manufacturing resources are controlled to deliver the unit-processes expected by the high-level control functions. High-level manufacturing control is concerned with co-ordinating the available manufacturing resource to make the desired numbers of types of products. Recently, it has been used to automatically and dynamically adapt changes at the physical machine level of control, which has been implemented on a real-time Java platform (Brennan *et al.* 2002).

The Product-Resource-Order-Staff Architecture (PROSA) developed at PMAKULeuven (Van Brussel *et al.*, 1998; Valkenaers *et al.* 1999) offered a promising solution to the problem of enforcing global objectives. Overall, the architectures used a centralised scheduler to take advantage of global information as

well as to allow the control system autonomously to enhance the advised schedule when it is suboptimal.

Another alternative architecture developed at the University of Calgary emphasised local decisions as well as a globally optimal solution. This is the market-based approach, known as the Metamorph I architecture (Maturana and Norrie, 1996). The general philosophy of this approach is that a continuously changing environment needs an adaptable and continuously evolving control system. It has been extended into Metamorph II (Shen *et al.* 2000) for developing more flexible, modular, scalable and dynamic systems.

Advantages of the hybrid control architecture can include:

a) All the advantages of hierarchical control.
b) Ability of local systems to have local autonomy. Borrowing from the features of heterarchical architecture, local "intelligence" has been increased to cope with unexpected conditions.
c) Ability to off-load some linkage tasks to local controllers. Since there is no strict master-slave relation, tasks of Local controllers can be re-adjusted without affecting higher-level controllers.

Disadvantages include:

1. Most of the disadvantages of hierarchical control.
2. Increased difficulty of control system design. Because of the increase of local autonomy, system functions become very difficult to allocate to appropriate control entities.
3. Requiring a higher network capacity. It is required by the increase in communication between controllers.

2.4.2 Architecture Description Language

Whilst architecture or architecture style is a strategic selection to help meet system requirements completely and efficiently, the architecture description language is the means to help to describe a specific system architecture design in order to meet

system requirements. Sometimes, it is referred to as a system modelling language and/or modelling method. However, unlike detailed design or implementation, architecture design concerns the conceptual aspect of the system design and evaluation. Normally, different modelling languages represent different system design philosophies, and relate to different underlying design methods. A selection of modelling languages that have been widely used in control and/or information system engineering is outlined below.

2.4.2.1 IDEF

IDEF (Integration Definition for Function Modelling) is a public domain USA Air Force developed method, which has been adopted by the Department of Defence (DoD) Corporate Information Management System. Originally developed by the USAF, IDEF is a graphical way to document processes. The original IDEF methods were developed for the purpose of enhancing communication among people who needed to decide how their existing systems were to be integrated (Underdown and Deese 1995).

While IDEF includes six sets of languages, the most utilised one is IDEF0. IDEF0 is derived from a well-established graphical language, the Structured Analysis and Design Technique (SADT) (Maarssen and McGowan 1981), which is considered as a method designed to model the decisions, actions, and activities of an organisation or system. Since it was developed during the early 1970s by Softtech (Colquhoun *et al.* 1993), it has been widely used, even today many companies, e.g. Lockheed-Martin, General Motors, Rockwell International, are still using IDEF for representing their processes (Zakarian and Kusiak 2001). Whilst similar in nature to the data flow diagram, the IDEF0 diagram distinguishes between data flow and control flow by the placement of the arrow on the rectangular boxes (Yourdon 1989). The underlying concept is based on a functional decomposition approach whereby a system is hierarchically decomposed into subsystems (each subsystem consists of one or more tasks). Therefore, an IDEF0 model consists of a hierarchy of related diagrams. Each diagram is based on a diagonal row of boxes connected by a network of arrows (Mayer *et al.* 1994). This makes it suitable for defining and documenting top down

hierarchical and modular systems. Based on the functional decomposition approach, the control system needs to be segmented into subsystems, where ideally there is a high level of cohesion within a subsystem and weak coupling between subsystems (Kavi and Yang 1989). In terms of the degree of coupling between modules (and tasks within a module), it will determine the amount of concurrency and govern delays due to synchronisation.

Since the mid 1970s, IDEF0 has been applied by various researchers. For instance, an IDEF0 model was proposed to support the development of rules for production planning and control of a cell controller (Pandy 1992). Gong and Lin (Gong and Lin 1994) have demonstrated it as a starting tool for control determination, classification, and allocation. Bauer *et al.* (Bauer *et al.* 1991) have applied it to present the shop floor control functions and related information system. The IDEF0 decomposition process has also been used as a basis for building up a manufacturing information model, as given in Kim *et al.* (Kim *et al.* 1993).

However, it has been shown (Dotan and Ben-Arieh 1991) that this method is inadequate in machine control system modelling because: the information encoded is abstract and may not necessarily correspond to physical or logical items, relationships between data are not well defined, the functionality of each activity is not specified; and no sense of time or sequence of operations can be encoded in the model.

2.4.2.2 UML

Unified Modelling Language (UML) is a modelling language for specifying, visualising, constructing, and documenting the artefacts of a system-intensive process. It represents another important branch of modelling methods, namely Object-Oriented Analysis and Design (OOAD). UML emerged from the unification that occurred in the 1990s following the "method wars" of the 1980s and 1990s. Among these methods, the most significant ones are Booch (Booch 1994), Coad and Yourdon (Coad and Yourdon 1991), OMT (Rumbaugh *et al.*, 1991), Martin Odell (Martin and Odell 1995), OOSE (Jacobson *et al.* 1992), and RDD (Wirfs-Brock 1990). UML is considered a third-generation object-oriented modelling language. Although UML

evolved primarily from various second-generation object-oriented methods (at the notation level), its scope extended its usability far beyond its predecessors. Furthermore, the experience gained, through experimentation and gradual adoption of the standard, reveal its true potential and enable organisations to realise its full benefits.

UML is an evolutionary general-purpose, broadly applicable, de-facto industry-standard modelling language with associated tool-set support. It could apply to a multitude of different systems, domains and processes (Sinan 1998).

- As a general-purpose modelling language, it focuses on a core set of concepts for acquiring, sharing and utilising knowledge coupled with extensibility mechanisms.
- As a broadly applicable modelling language, it may be applied to different types of systems (software and non-software), domains (business and software) and methods or processes.
- As a supported modelling language, tools are readily available to support the application of the language to specify, visualise, construct and document systems.
- As a de-facto industry-standard modelling language, it is not proprietary but open and fully extensible.

Because of these UML features, various researchers have adopted it to facilitate machine control system development. Lin and Ming (Lin and Ming 2001) use UML for the development of a holon-based manufacturing control system (MCS), which combines concepts from object-oriented analysis with the software architecture design and software implementation. It has also been used to model and analyse the ventilation process of a road tunnel and its control system in Prague, which is considered as a complex and heterogeneous system (Kerckhoffs and Snorek 2001). Fernandez *et al.* (2001) proposes extensions in order to enable UML to model and analyse real-time systems and he also gives a realistic example of an automated passenger train system. In addition, UML-RT is an extension of UML for modelling embedded reactive and real-time software systems (Fischer *et al.* 2001). Its particular focus lies in system descriptions on the architectural level, defining the overall system structure. Combining with the formal method CSP-OZ, UML-RT structure diagrams

are given the formal semantics to facilitate real-time system development. Furthermore, UML cannot only be used in software development, but can also be used in low-level hardware development. Diedrich and Neumann (Diedrich and Neumann 1998) use UML to specify the device model and they also give a comparison with existing device description languages. As they point out, UML can improve the integration of the whole control system development.

2.4.2.3 SDL

SDL is a Specification and Description Language standardised by the ITU (International Telecommunication Union) as standard Z.100 (Færgemand and Olsen 1994). Since the first version of the language was issued in 1976, the language has been evolving through 1980, 1984, 1988 until 1992, when object oriented features were included in the language (Ellsberger *et al.* 1997). SDL has been widely used to specify and validate communications protocol and communications systems. The method associated with SDL is considered to be a combination of object-oriented method and structured method, which uses an object-oriented design method at high level and describes data flow at low level.

SDL is intended for the description of complex, event-driven, real-time, and communicating systems. However, it is not only directed specifically at describing telecommunications services, but also it is a general-purpose description language for communication systems. For example, it has been used in general software design and real-time systems. It has also evolved in studies of hardware-software co-design (Peeters *et al.* 1995; Levin *et al.* 1996). SDL features a formal definition, that is rules that formally define the semantics behind each symbol and concept. SDL's formality enforces precision during specification and provides support for analysis and verification. SDL also supports dynamic features that are more software oriented. This high-level language improves productivity of the design process by letting the designer concentrate on the application problem instead of dealing with low-level programming issues.

Nowadays, control and telecom engineers are faced with a similar challenge, which is to design highly distributed control systems with complex interactions. Based on this consideration, SDL can serve both domains. Camus and Le Sergent (Camus and Le Sergent 2001) explored an approach to combine two complementary formal methods: SCADE/Lustre from the Control Engineering domain, and SDL, from the Telecom domain, in order to improve the distributed control system solution. Because it originates from the Telecom domain, it is good for the low-level control system such as an embedded control system (Jahnke 2001). Furthermore, it can also be employed to assist the high-level control system development, such as the whole control system for a car factory (Niere and Zundorf 1999; Sousa and Putnik 1999).

However, SDL lacks a capability to capture system requirements. This is the reason that it is recommended to combine it with UML during the development process (Telelogic 1998). From an object-oriented development point view, it also lacks a strong and syntactically clean distinction between classes/types and interfaces; it lacks an object-oriented data model with polymorphic properties and reference signal parameters; the complexity of the language is due to a large number of overlapping concepts.

2.4.2.4 Petri-Nets

Petri-Nets are a well-known method for modelling and analysing event-based systems, which are characterised as being concurrent and asynchronous. It is regarded as a formal mathematical method, which has been successfully applied in modelling multi-event synchronisation. Since it was proposed by CA Petri in 1962 (Petri 1962), it has been used in many areas including simulation, performance evaluation, analysis, real-time control, and modelling of communication systems, etc. (Petri 1980).

The involvement of Petri nets applications in industrial engineering systems, particularly manufacturing systems, started in the early 1980s. Since then, the Petri nets methodology and its applications in automated manufacturing systems have been entirely explored and several results can be observed (Desrochers and Robert 1995). In an industrial manufacturing system, products are typically composed of discrete

parts and behaviours to transform raw materials into end goods are dominated by discrete event activities, which is particularly suitable for Petri nets. Recently, Lee and Park (Lee and Park 1993) have described the concepts of "object" and "message-passing" in object-oriented programming to develop object-oriented Petri nets (Opnets) models to increase the maintainability and reusability of objects used in Petri net modelling. PN models can have time parameters attached to them to indicate temporal capabilities, in a manner appropriate to activity modelling in respect of manufacturing machines (Abdallah *et al.* 2002). It also gains increasing recognition in industry as a practical tool (Zha 2002). The advantages of PN include: ease of understanding and readability (Liu and Wu 1993); graphical representation suits the description of distributed and concurrent systems (Wang and Wang 1995); allowing analytical validation and direct implementation techniques suitable for rapid prototyping based on real-time interpretation (Roux *et al.* 2001). The mathematical definitions from PN schema (Castillo *et al.* 1999) enable computer programs to be produced for machine control software applications.

However, the effort involved in the design and construction of Petri net based models is high for complex systems due to the fact that Petri nets modelling is system dependent and lacks certain properties including the modular programming, reusability and maintainability that are commonly required in machine control systems.

2.4.2.5 Summary

It is hard to claim the delivery of a final solution or a universal tool. Thus, the challenge is not to develop or find an all-capable universal tool, but the right tool for the job. The same principle can be applied to the selection of the description language. Based on the selection criteria, the comparison of reviewed description languages is summarised in table 2.1.

The primary objective in conceptual design is to capture the requirements in developing a machine control system, in order to drive system development in the right direction. This can only be achieved if the requirements are specified and

organized within a well-defined framework. As table 2.1 shows, only UML and IDEF are capable of assisting the engineering process to gather requirements.

To cope with system complexity, an essential role for system design is to distribute responsibilities into different aspects of the identified problem space, in the form of distinct modules that conform to a pre-defined functional architecture. With the abstractions, we can concentrate on the limit system temporal and spatial structures, which in turn determine the essential properties of the system. Typically, for a description language, it is necessary to support a hierarchical decomposition approach in order that complex tasks can be broken down into solvable parts and formalised solutions. In this aspect, IDEF0, UML and SDL can be used for a hierarchical structure.

One important demand to achieve agility in a machine control system is to cope with unpredictable change. Long-term forecasts are increasingly hard to make in an ever faster changing environment. The component-based development approach can readily achieve this as it makes change become more of a permanent internal process rather than an external factor that the control system has to adapt to. UML is the one that inherently supports component-based development. In addition, it does not only focus on software component design, but also empowers hardware component development.

Although it is primarily concerned in describing static high-level system structure, conceptual control system design is inevitable to engage in the analysis and description of system dynamic behaviours. In fact, machine control systems consist of a substantial amount of synchronous and asynchronous tasks. It is necessary to explicitly analyse and present these tasks supported by the description language, which is lacking in IDEF0. Moreover, at the conceptual design stage, the main task is focused on requirement collection and idea representation. Thus, whether the model is executable or not is unimportant for conceptual design, although it brings great benefits in term of the intuitive evaluation of the design.

Overall, the results lead the author to conclude that UML is the one, which would be most appropriate at the present time for the design of a control architecture. Since

UML is a general, semi-formal language, and is not specified for control systems, it needs some enhancement in certain aspects, so that it can serve more efficiently for control system design.

	IDEF0	UML	SDL	Petri Nets
Hierarchical Structure	■	■	■	
Represent dynamic behaviour		■	■	■
Capturing control requirement	■	■		
Well-known in control domain	■	■		■
Executable			■	■
Component-based development		■		
Graphical representation	■	■	■	■
Hardware description		■	■	

■ Yes

Table 2.1 Description Languages Comparison Table

2.5 Control System Implementation Technologies

In this research work, the implementation of conceptual design results relies mainly on technologies of control software, which involve underlying software architectural mechanism and realisation languages. Thus, three popular standards of software component architecture are reviewed, and the implementation languages are described in this section from a control system perspective.

2.5.1 Software Component Infrastructures

Software component infrastructures have emerged as a standard design paradigm in many areas of application development. JavaBean (Englander 1997) is a component standard for building Java-based desktop applications. COM/DCOM (Sessions 1997) is Microsoft's component model that is central to their application interoperability. CORBA (OMG 1995) is an open platform component infrastructure provided by

Object Management Group (OMG). These are typical component infrastructures for a distributed control system. Kang *et al.* (2001) have implemented these three types of distributed control system and compared their performance. The following will briefly introduce them, and separately discuss their advantages and disadvantages.

2.5.1.1 COM/DCOM

COM (Component Object Model) is Microsoft's component standard that forms the basis for interoperability among all Windows-based applications. Microsoft has developed a distributed version of COM, called DCOM, which targets networked Windows systems, which is often called '*COM on the wire*', and supports remote objects by running on a protocol called the Object Remote Procedure Call (ORPC).

COM is a binary compatibility specification and associated implementation that allows clients to invoke services provided by COM-compliant components (COM objects). Services implemented by COM objects are exposed through a set of interfaces that represent the only point of contact between clients and the object. COM defines a binary structure for the interface between the client and the object. This binary structure, which is a virtual function tables (or vtable) layout, provides the basis for interoperability between software components written in arbitrary languages. Interfaces that can be operated have to implement IUnknown interface, which is the base interface of all COM interfaces. As long as a compiler can reduce language structures down to this binary representation, the implementation language for clients and COM objects does not matter - the point of contact is the run-time binary representation. Thus, COM objects and clients can be coded in any language that supports Microsoft's COM binary structure. COM objects and interfaces are specified using Microsoft Interface Definition Language (MIDL), an extension of the DCE Interface Definition Language standard. To avoid name collisions, each object and interface must have a unique identifier.

Every COM object runs inside a server. A single server can support multiple COM/DCOM objects. There are three ways, in which a client can access COM/DCOM objects provided by a server:

- In-process server: The client can link directly to a library containing the server. The client and server execute in the same process. Communication is accomplished through function calls.
- Local Object Proxy: The client can access a server running in a different process but on the same machine through an inter-process communication mechanism. This mechanism is actually a lightweight Remote Procedure Call (RPC).
- Remote Object Proxy: The client can access a remote server running on another machine. The network communication between client and server is accomplished through DCE RPC.

An important aspect in COM is that objects have no identity, that is a client can ask for a COM object of some type, but not for a particular object. Every time that COM is asked for a COM object, a new instance is returned. The main advantage of this policy is that COM implementations can pool COM objects and return these pooled objects to requesting clients. Whenever a client has finished using an object the instance is returned to the pool.

As Window DNA for Manufacturing and Microsoft Windows 2000, NT, and CE have been adopted in machine control systems, COM and its extended technologies such as DCOM and OPC, have been considered as a solution for machine control system. Additionally, they have started to emerge in other OS platforms such Solaris (Microsoft, 2000). The NS2000 SCADA system consisted of a substation automation database, network and graphic interface for system configuration, which was based on DCOM (Ding *et al.* 2001). A multi-agent-based distributed control system for flexible production systems was also reported, which was built on embedded control agents (Schoop *et al.* 2001a). The agent components were implemented as Windows NT services and logic control programs interfaced via DCOM and Ethernet. The results of the application in an industrial flexible production system were used to show the effectiveness of the proposed approach. Furthermore, DCOM and OPC together with fieldbus systems can provide solutions for conventional PLC and CNC control equipment as well as Industrial PCs (Schoop *et al.* 2001b; Xie *et al.* 2002)

2.5.1.2 JavaBean/RMI

Java is an object-oriented programming language developed by a small team of people headed by James Gosling at Sun Microsystems development began in 1991. It was originally intended for use in programming consumer devices, but when the explosion of interest in the Internet began in 1995 it became clear that Java was an ideal programming language for Internet applications (Van Hoff 1996). When they are embedded in a Web page, Java programs are called "applets." Applets, in conjunction with JavaBeans provide a developer the flexibility to develop a more sophisticated user interface on a Web page. Java applets are the dominant player in client side Internet computing. However, the server side computing was considered a stronghold of better performance languages such as C++ or script languages such as PERL. This situation is changing with the release of Java 2 Enterprise Edition (J2EE). J2EE is a new Java platform specifically designed to address the needs of enterprise server side computing. Java is also re-addressing its original purpose (consumer devices) through Jini connection technology. Jini enables devices to work together without the burden of setting up networks, loading drivers and so on. Jini devices such as TVs, DVDs and cameras will be able to self-install, self-organize into communities, self-configure, and self-diagnose.

A client accesses the JavaBean object supported mainly by the object serialisation service and the remote method invocation (RMI) service. An object is handled via references of interface type. Interfaces that can be accessed have to be derived from java.rmi.Remote. If an argument is of remote interface type, the reference will be passed. In all other cases, passing is by value, which is the argument is serialised at the call site and deserialised before invoking the remote interface.

Object identity is affected by Java RMI, as a result of its model of implementing remote references. If a remote interface reference is passed around, proxy objects are created on remote sites. A reference to a remote interface, once passed in a remote method invocation, is thus not a reference to the remote object but a reference to the local proxy of that object.

The Java distribution model extends garbage collection as well fully-distributed garbage collection is supported, based on careful bookkeeping of which objects may have remote references to them. Distributed garbage collection is the most outstanding feature of Java RMI, when compared with any other approach in the mainstream today.

Currently, Java applications for industries are limited to remote operations such as maintenance, diagnoses and command executions from a desktop browser via an intranet or the Internet. In particular, Inoue *et al.* (2002) have developed a dynamic program sending and automatic starting architecture for flexibly responding to changes in requirements from factory, chemical plants, or remote-side operators. By using this architecture, a remote-side operator is able to select best match programs whenever they are needed and dynamically send and start them. However, it is still in its infancy in manufacturing industry for performance and reliability reasons. Some current standardization activities, such as Real-time Java and the Java-based Industrial Monitoring Framework, could potentially benefit future machine control systems (Hoshi 1999).

2.5.1.3 CORBA

The Common Object Request Broker Architecture (CORBA) is a specification of a standard architecture for object request brokers (ORBs). A standard architecture allows vendors to develop ORB products that support application portability and interoperability across different programming languages, hardware platforms, operating systems and ORB implementations.

The CORBA specification was developed by the Object Management Group (OMG), an industry group with over six hundred member companies representing computer manufacturers, independent software vendors and a variety of government and academic organisations. Thus, CORBA specified an industry/consortium standard, not a "formal" standard in the IEEE/ANSI/ISO sense of the term. The initial CORBA specification emerged in 1992. Since then, the CORBA specification has undergone

significant revision, with the latest major revision (CORBA v2.0) released in July 1996.

CORBA ORBs are middleware mechanisms, as are all ORBs. CORBA can be thought of as a generalisation of a remote procedure call (RPC) that includes a number of refinements of RPC, including:

- A more abstract and powerful interface definition language;
- Direct support for a variety of object-oriented concepts;
- A variety of other improvements and generalisations of the more primitive RPC.

It is impossible to understand CORBA without appreciating its role in the Object Management Architecture (OMA). The OMA is itself a specification (actually, a collection of related specifications) that defines a broad range of services for building distributed applications. The OMA goes far beyond RPC in scope and complexity. The distinction between CORBA and the OMA is an important one because many services one might expect to find in a middleware product such as CORBA (e.g., naming, transaction, and asynchronous event management services) are actually specified as services in the OMA. For reference, the OMA reference architecture encompasses both the ORB and remote service/object

OMA services are partitioned into three categories: CORBAServices, CORBAFacilities, and ApplicationObjects. The ORB (whose details are specified by CORBA) is a communication infrastructure through which applications access these services, and through which objects interact with each other. Every object interface inherits from CORBA.Object. CORBAServices, CORBAFacilities and ApplicationObjects define different categories of objects in the OMA; these objects (more accurately object *types*) define a range of functionalities needed to support the development of distributed software systems.

An important point to note is that CORBA specifies that clients and object implementations can be written in different programming languages and execute on different computer hardware architectures and different operating systems and that clients and object implementations cannot detect any of these details about each other.

Put another way, the IDL interface completely defines the interface between clients and objects; all other details about objects (such as their implementation language and location) can be made "transparent".

CORBA technology has attracted increasing attention from academics and industry. It has been used in a wide area from manufacturing systems down to numerical controllers (Jiao and Yu 2001; Raymond *et al.* 2001). It has also been considered as a technology to integrate manufacturing systems (Curto *et al.* 2001).

2.5.1.4 Comparison

The architectures of COM/DCOM, JavaBean/RMI, and CORBA provide mechanisms for transparent invocation and accessing of objects. Though the mechanisms that they employ to achieve may be different, the approach each of them takes is more or less similar.

Table 2.2 gives a detailed comparison.

	COM/DCOM	*JavaBean/RMI*	*CORBA*
OS platform support	Any platform as long as there is a COM Service implementation for that platform	Any platform as long as Java Virtual Machine (JVM) is available	Any platform
Language	Any language	Java	Any language
Interface Description	MIDL/IDL	Java	IDL
Interface Identification	Universally Unique Identifiers	Logical name	Logical name
Protocol	Uses the Object Remote Procedure Call (ORPC)	Uses the Internet Inter-ORB Protocol (IIOP)	Uses the Java Remote Method Protocol (JRMP)
Object Handle	Interface pointer	Object reference	Object Reference
Locating and Activating Object	Service Control Manage (SCM)	ORB for locating Object Adapter for activating	JVM
Integration method	Binary	Interface	Java Interface
Distributed garbage collection	Attempt	Not attempt	Attempt

Table 2.2 Comparisons of COM/DCOM, JavaBean/RMI, and CORBA

For machine control systems, the performance of systems is a primary concern. Obviously, Java is not a good candidate. Because the Java run-time environment relies on the Virtual Machine, it makes Java inefficient for this purpose. Although there are some approaches providing Java real-time capability (Puschner and Wellings 2001), there is still a long way to go for it to become a standard. Furthermore, Java only provides an integration platform for itself, no other languages are supported.

Comparing COM/DCOM to CORBA, it is tightly integrated with the operation systems. In other words, the Windows family OS embeds the COM/DCOM mechanism, in particular, Windows NT and subsequent new releases are built on COM/DCOM. This gives better performance than CORBA. The advantage of COM/DCOM is more evident when objects are deployed in the same computer or the same memory address, which is also a normal technique used to meet time critical requirements. Recently, an established high-performance CORBA working group (Marotta *et al.* 2001) has been established, which may eventually address a subset of performance concerns, but there is still long way to go. Although CORBA has been standardised, a CORBA standard for machines is still under development (OMG 1998). In comparison, OPC has been widely accepted by industry. Subsequently, OPC has been adopted in some real-time OS besides the Windows family OS. For example, OPC has been available in VxWork (Arthanari 2002).

The versioning problem is another factor that needs to be considered. As the table shows, CORBA was the logic name to identify the object. Thus, if a different version component is in the same server, it will cause problems. While COM/DCOM does not have this problem, because the UUID is used to identify the object, the UUID of the different version component will be different. Associated with this, another benefit of COM/DCOM is that it allows updating of a component in an application without recompilation, re-linking, or even restarting.

2.5.2 Programming Languages for Component Implementation

Despite the declared independence of components from programming languages, components still need to be constructed in some way. Component construction itself can be performed using almost arbitrary programming languages, as long as the language and its implementation support the particular component standard's interface conventions. There are well over 100 object-oriented, object-based, and component-oriented languages. Most of them can achieve a component-based development strategy. The following is a selection of some popular programming languages.

In the engineering field, the most popular language may be C. The main reason is that it is possible to do anything at any level of the operating system. Although C can achieve component development, it does not fully exploit the power of component development. That is the reason that programmers appreciate an object-oriented language. One approach to efficient object-oriented programming that has been popular is to extend an existing language to include object-oriented features. C serves as an example of being extended in this way. There are two essential ways, in which this can be done, exemplified by the languages Objective-C and C++. Objective-C (Cox and Novobiski 1991) extended C by including one additional data type in the form of a C structure within a function library with the functionality of Smalltalk. AT& T's C++ (Stroustrup 1997) on the other hand actually changed the C compiler and extended the syntax with a new primitive data type for class. C++ is considered as a compromise between the object-oriented ideal and pragmatism.

Java is another popular language. When Java emerged, it took industry by storm. It was a web-enabled language supporting security and concurrency, which are features missing in C and C extensions, so that small Java programs could be delivered as applets to run in a browser such as Explorer or Netscape Navigator. Fully-fledged applications could also be written. Its syntax resembled C++ but it is a pure object-oriented language, has automatic memory management and discourages the use of pointers; thereby being much safer. It was put in the public domain and supported by a powerful vendor, Sun. Java comes with its own object request broker technology, RMI. This allows applications to call methods executing in other Java applications across a network. It has also incorporated CORBA-compliance since version 2.

While Java stems from C, the Component Pascal language has its roots in another major language, Pascal. This language has been used for an extremely wide spectrum of tasks. It can be used for programming low-level systems such as device drivers, embedded systems, real-time programming, operating system kernels, interrupt handlers *etc.* It can also provide complex graphical user interfaces, compound document systems, the BlackBox component framework and even high-level scripting. While Component Pascal can be implemented efficiently, it is a safe language. A simple proof of concept is that Component Pascal can be compiled to Java byte code.

There are some simple languages named scripting languages such as Javascript. Although they are classical compiled object-oriented languages, they support the use of most important techniques typical in OOP. Moreover, its loose typing offers new possibilities like dynamic updating and/or addition of properties and methods not available in classical OOLs. Inheritance programmed explicitly by the user is very flexible (Sklenar 2000). It is also a trend to extend them to wide areas such as concurrency computing (Yan *et al.* 2001).

The above languages have been developed for software engineering purposes. For control engineering, IEC 1131-3 (also referred to as its full title IEC 61131-3) has emerged as a language for control software development (IEC 1993). IEC 1131-3 is really the only internationally recognised standard language for industrial automation control. It harmonizes the way people look at industrial control by standardising the programming interface. It includes the definition of Sequential Function Charts (SFC), two text-based languages: Instruction List (IL) and Structured Text (ST) and two graphical languages: Ladder Diagram (LD) and Function Block Diagram (FBD). In particular, SFC is specified as a means to describe the sequential behaviour of a control programme as an aid to structure the internal organisation of a programme. Furthermore, it also supports functions written in 'C'. Via modularisation and declaration of variables, each program is structured, increasing its re-usability. In addition, IEC 1131-3 can be used to configure the structures of a control system. Initially, it is intended to provide a standard, so that users can move between different brands and types of control with very little training and exchange applications with a minimum of effort. In a recent development, it has integrated fuzzy control applications to IEC 1131-3 languages (IEC 2000) and standardised function blocks for motion control (PLCopen 2001). It also encourages well-structured, 'top-down' or 'bottom-up' program development. Since the concept of the soft Programmable Logic Controller (PLC) emerged, which is more strongly associated with software rather than PLC hardware, it has been adopted as a general language (Bonfe and Fantuzzi 2000). Although it is not an object-oriented language, several approaches have been applied to enable component-based development (Ohman *et al.* 1998). There is another approach proposed by the author, which will be discussed in chapter 5.

One point is necessary to note. Comparing the languages, the diagrammatic syntax of IEC 1131-3 consists of circuit (i.e. box-and-wire) diagrams, emphasising a data-flow view, and variants of Petri net diagrams, suited to a control-flow view. In particular, SFC not only can be used as a tool to express ideas, but also can be executed. It gives the advantage during the entire development phases from detail design to final implementation (Wegrzyn *et al.* 1998).

2.6 Summary

This chapter has reviewed technical and research literature related to machine control systems. In this context, it reviewed various approaches to control system analysis and design, in particular component-based development has been featured. For control system conceptual design, various architectures have been considered to provide a rationale for selecting the system architecture. Several architecture description languages have been evaluated. Furthermore, several underlying component architecture standards and implementation languages have been compared.

Chapter 3 Control System Design and VIR-ENG

3.1 Introduction

The research work described in this thesis constitutes an integral part of the Framework IV ESPRIT project 25444 - VIR-ENG, "Integrated Design, Simulation and Distributed Control of Agile Manufacturing Modular Machinery". This chapter will highlight the key features of "VIR-ENG" and the role of machine control system design in the context of the complete research programme. It is envisaged that the line of arguments developed in this chapter will serve as the roadmap for the research findings detailed in the chapters to follow.

Before focusing on machine control system design, the VIR-ENG background is outlined. Based on the VIR-ENG top-level model, the roles of the Control System Design Environment (CSDE) are identified and the conceptual solutions and associated tools are proposed to address control system design issues. The design process for control systems is then described.

3.2 Overview of VIR-ENG

3.2.1 VIR-ENG Objective

The aim of VIR-ENG is to develop highly integrated design, simulation and distributed control environments for building agile modular manufacturing machine systems which offer the inherent capacity to allow rapid response to product changes and feature variants (Pu and Moore 1998). In adopting a component-based design and integration paradigm, VIR-ENG seeks to facilitate functional requirements from different 'user' levels and perspectives, namely: (i) to close the gap between the machine system design and control system design; (ii) to develop a mapping between the 'virtual' world and the 'real' world and (iii) to build coherent links between the design environment and the run-time environment of automation / machine systems.

3.2.2 VIR-ENG Environment Reference Model

The VIR-ENG research project creates an integrated design, simulation, operation, and maintenance environment for realising agile manufacturing machine systems. Three perspectives have been identified that encapsulate the major constituents of the philosophy, which have been adopted by the VIR-ENG project, namely the design view, the simulation view and the control view, as shown in figure 3.1.

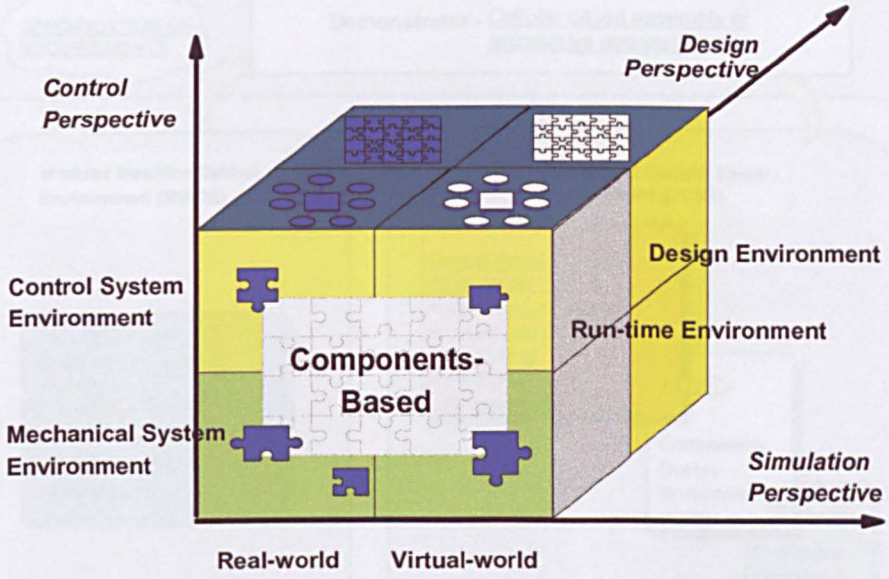

Figure 3.1 Reference Model for the Integrated Environments

The control perspective encapsulates the functional levels within a manufacturing machine, namely the 'control system' and 'mechanical system'. The mechanical system could be configured mainly from modular machinery elements; the control system could be implemented in distributed forms as control components utilising recent developments in fieldbus technology, smart sensing techniques, and software component technology. The simulation perspective encapsulates the virtual perspective of the design and run-time processes of both the modular machine design environment and the distributed control system environment. The introduction of the virtual process and component enables concurrent design processes to be established and coherently integrated. Furthermore, on-line reconfiguration of the machine/system can be carried out, resulting in reduced lead-time, machine downtime and product changeover time. The design perspective encapsulates the design and run-

time attributes of developing manufacturing machine systems. The integration of these three perspectives form the essence of the VIR-ENG research project to realise 'customised design', implementation and life cycle support of re-configurable modular manufacturing systems.

3.2.3 VIR-ENG Environments

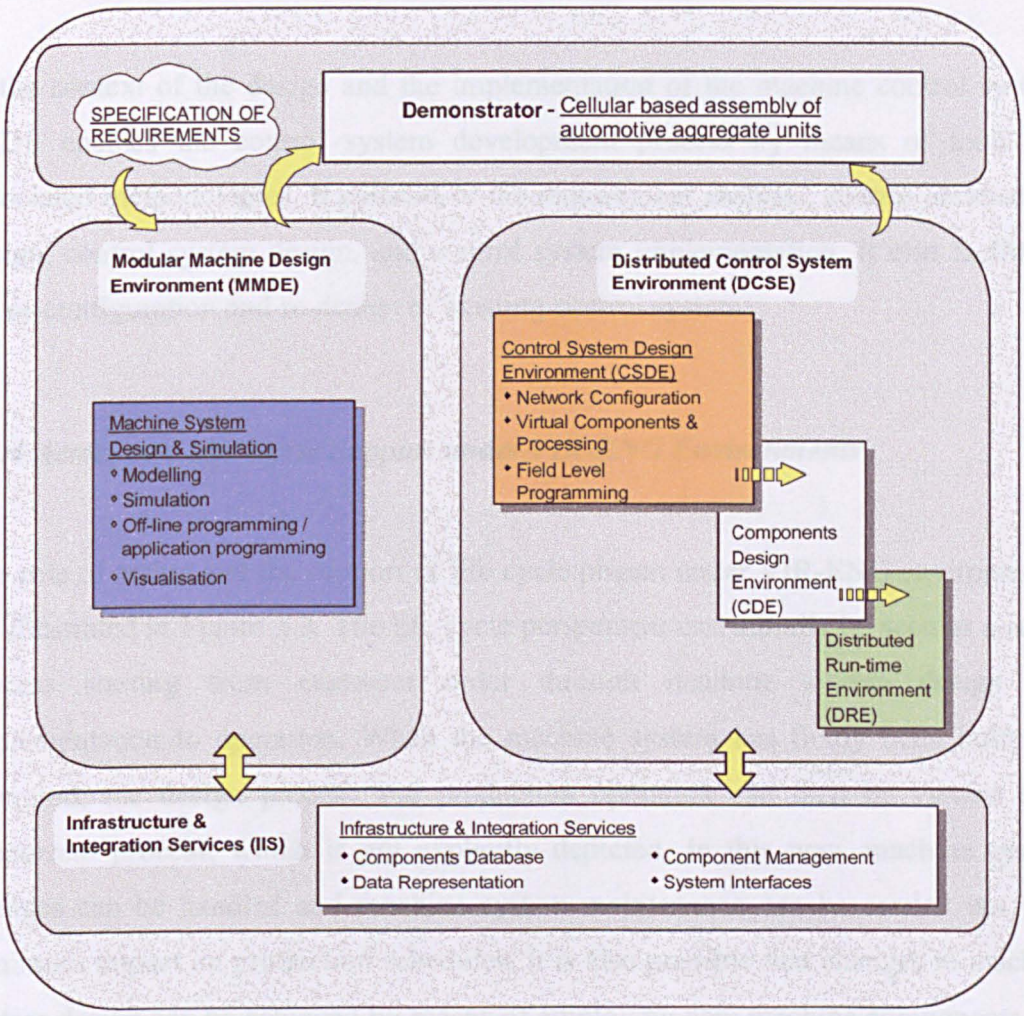

Figure 3.2 The Partition of the VIR-ENG Environment

There are two main integrated environments that are provided by VIR-ENG for the design and realisation of modular manufacturing machine systems shown in Figure 3.2. These are the Modular Machine Design Environment (MMDE), which supports interactive visualisation, modelling and simulation, and the Distributed Control System Environment (DCSE) for the design and run-time operation of control

systems. Within DCSE, there are three main environments, namely Control System Design Environment (CSDE), Component Design Environment (CDE), and Distributed Run-time Environment (DRE). CDE is mainly concerned with the abstraction of hardware functions to meaningful control functions and DRE mainly focuses on supporting the run-time operation of machines. CSDE primarily facilitates the machine control system design and implementation. An associated Infrastructure and Integration Services (IIS) was developed to facilitate coherent support for MMDE and DCSE.

In the context of the design and the implementation of the machine control system, CSDE enables the control system development process by means of tools and associated methodologies. It consists of the requirement analysis, control architecture design, control system design, and control system implementation. It also facilitates the re-configuration and re-design of existing control systems.

3.2.4 Actors and Life Cycle Support under VIR-ENG Environments

The role of actors and the support of life cycle phases under VIR-ENG environments are illustrated in Figure 3.3. The life cycle perspective can initially be seen as a serial process starting from customer order through machine system design and implementation to operation. When the machine system has firstly been built and delivered, the design process and production operation can then be viewed as a concurrent process, which is not explicitly depicted. In this way, machine system changes can be handled and machine system maintenance can be carried out with minimum impact on production schedules. It is also possible that changes to machine system design can be achieved by means of employing new machine components and by reconfiguring the machine system. Maintenance tasks can be carried out through replacing or upgrading flawed components. A key difference compared to the traditional life cycle approach is that explicit reconfiguration scenarios have been taken into account by design.

It is evident that this process requires an engineering team from different discipline backgrounds to facilitate every step forward in the machine system life cycle. These actors are also named in Figure 3.3. As a whole, the VIR-ENG environments support

multi-discipline cooperative design actions through inherent interactions between environments, which can be referred to as an agile development process. Furthermore, VIR-ENG environments are designed to support the entire machine system lifecycle, including the collection of use-phase operation data and the design/validation of new system configuration.

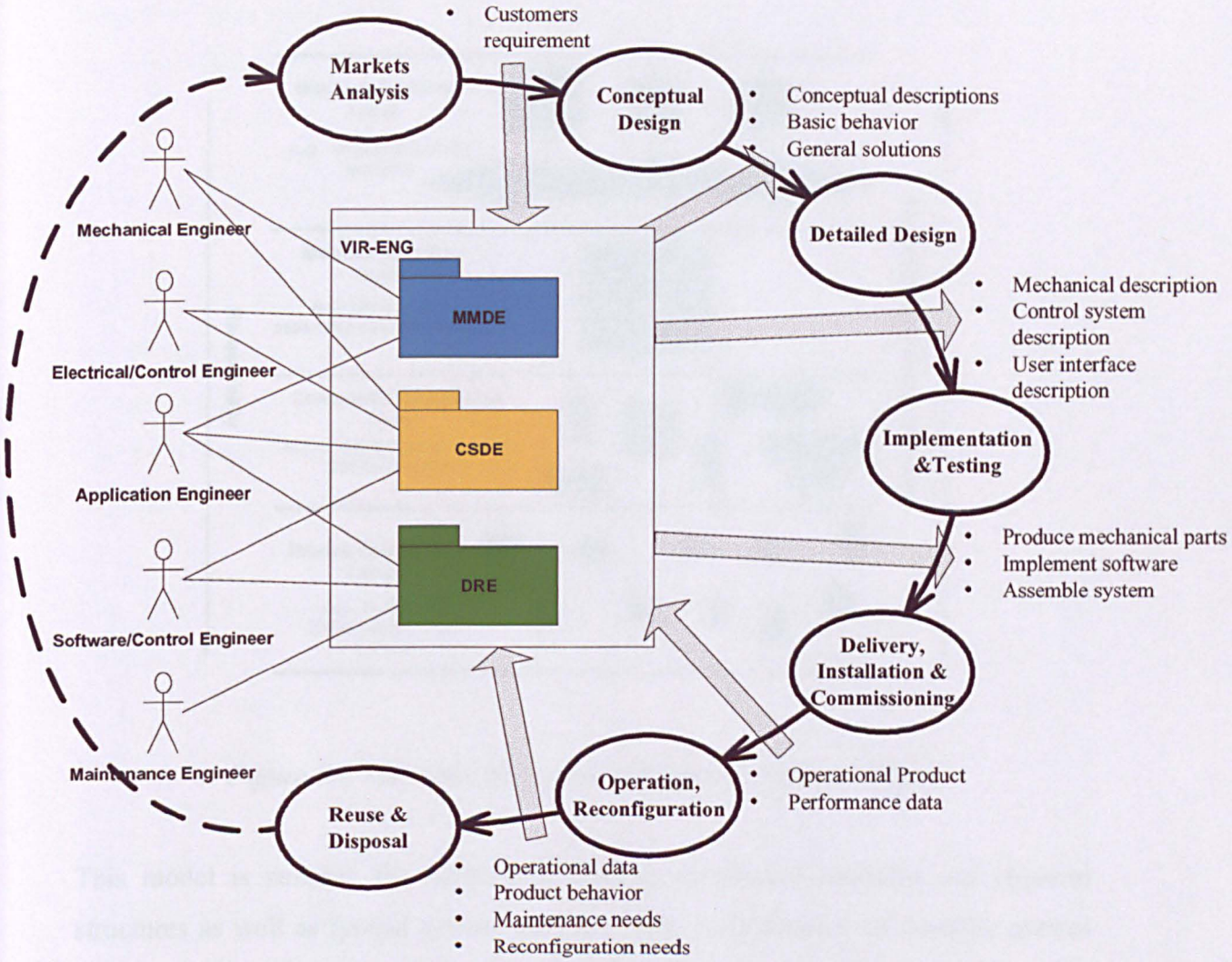

Figure 3.3 Actors, VIR-ENG Environments, and the Agile Machine System Lifecycle

3.2.5 VIR-ENG Manufacturing Machine System Model

In order to ensure the integrity of both MMDE and DCSE in analysing, designing and implementing the machine system, a four-layer model has been devised to serve as a reference model, as shown in figure 3.4. The four layers in the model are defined as follows:

4th Layer: Machine Systems

3rd Layer: Modular Machine

2nd Layer: Composite Components

1st Layer: Device Components

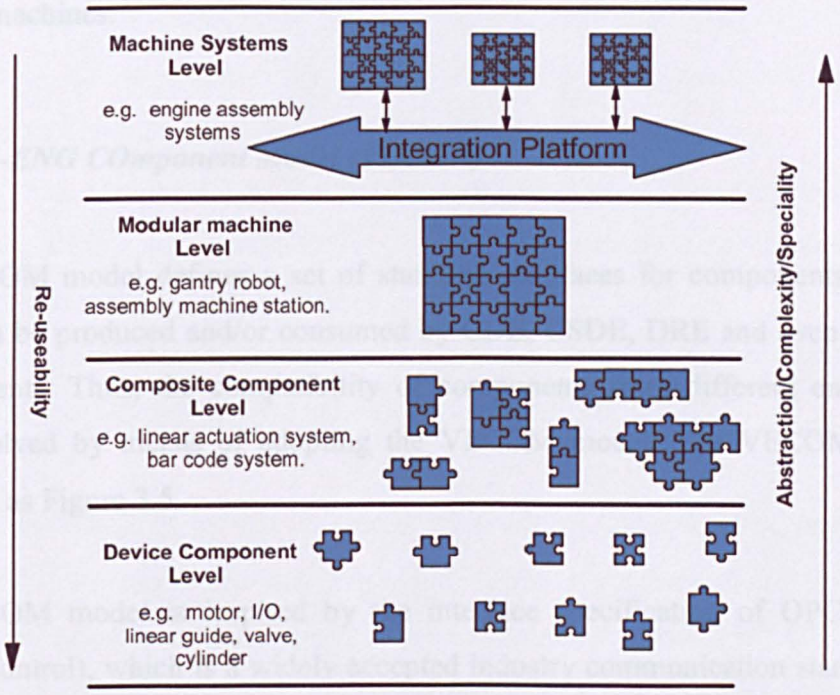

Figure 3.4 VIR-ENG Manufacturing Machine System Model

This model is suitable for partitioning typical mechanical modules and physical structures as well as typical control systems. With consideration of machine control system design, the Automated Manufacturing Research Facility (AMRF) model (Bunce 1988) can be used to interpret this model. The Device Components level, corresponding to the equipment level in AMRF, consists of individual resources such as the machine tools, sensors, and actuators, which control the detailed movements and operation of the equipment in achieving the tasks. The Composite Components level cannot be found a corresponding level in AMRF; its position lies between the equipment level and workstation level. In terms of complexity, it is more complicated than the equipment level, but it is not complex enough to achieve a complete task in the same way that the workstation level does. This level is important to facilitate

component development and reuse by grouping a number of device components to provide higher-level functionality. The Modular machine level corresponds to the workstation level, which executes the detailed operation sequence in order to complete a job despatched from the upper level. The cell level in AMRF has the same function as the machine systems level, which co-ordinates the modular machine operation, schedules the jobs specified from upper levels and despatches jobs to the modular machines.

3.2.6 VIR-ENG COmponent Model (VECOM)

The VECOM model defines a set of standard interfaces for components in DCSE, which can be produced and/or consumed by CDE, CSDE, DRE and even third-party environments. Thus, the compatibility of components from different environments can be solved by means of adopting the VECOM model. The VECOM model is illustrated as Figure 3.5.

The VECOM model is inspired by the interface specification of OPC (OLE for Process Control), which is a widely accepted industry communication standard (refer to 2.3.1). However, OPC is normally used as middleware to integrate different systems together, whereas VECOM is a general model for component-based development rather than just middleware. As such, some modifications and improvements have been incorporated into the VECOM model based on the OPC interface specification.

Based on OPC, VECOM utilises the interface of OPC for data access, which includes the Address space browsing interface (IOPCBrowseServerAddressSpace), the Data organising interface (IOPCServer), the Data access interface (IOPCSyncIO, IOPCASyncIO) and the Configuration persistence interface (IPersistFile). Although VECOM adopts the same interface item names as OPC, the usage is different. The result from Address space browsing interface return is a pointer or a functional call pointer, which is the data access address as in OPC. While synchronous data access is the interface for importing and exporting data, asynchronous data access deals with events and messages. Besides the OPC standard interfaces, VECOM adds in some

new interfaces, which are not defined in the OPC interface specification. The interfaces can be described as follows:

- Component configuration interface (IPropertyPage). This interface allows configuration functions to be built-in to the component and provides an interface for support tools to configure a component. Potentially, it provides the possibility for on-line configuration of a component.
- Alarm and Error handling interface (IErrorLookup). This is the interface providing the mechanism to inform the component consumers, so that the consumers can handle errors.
- Application Interface. An interface that provides some specific methods and/or services, which are related to the specific component, such as special motion functions, or vendor specific device information.

Although the VECOM model has extended the OPC interface, it is still compatible with OPC Servers by means of retaining the essential core of OPC.

Although the VECOM model has been adopted by CDE, CSDE, and DRE, the considerations to implement this model are different in these three environments. CDE is concerned with wrapping equipment device functions; CSDE focuses on encapsulating control functions; DRE embeds graphic interfaces in components for operation. Therefore, the component implementation structure is different, dependent upon the different environment needs. The implementation structure in CSDE is shown in figure 3.5. The control logic kernel encapsulates and executes assigned control functions and a real-time database responds to build a standard bi-direction communication tunnel between the kernel and the external consumer.

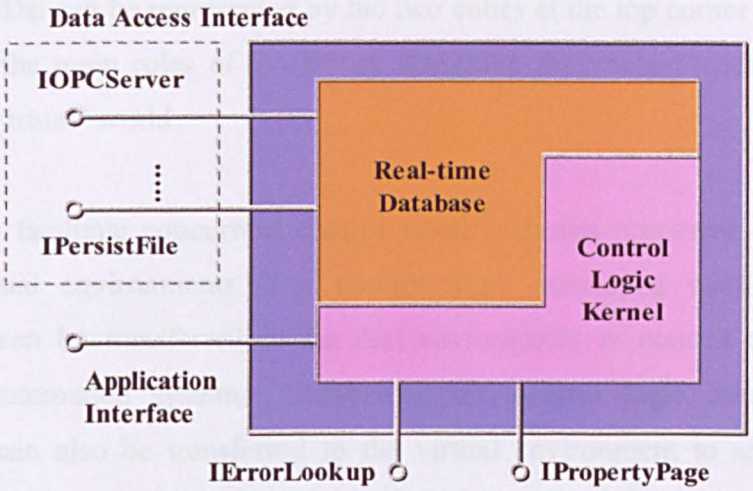

Figure 3.5 VECOM Model and its Implementation Structure in CSDE

3.3 Control System Design Environment (CSDE)

3.3.1 Responsibilities of CSDE

Modern machine systems of the type addressed by VIR-ENG consist of suitable combinations of mechatronics (mechanical and control system) elements, with application specific code determining how the particular machine behaves (Harrison *et al.* 2000). According to Philips (2000), the term 'mechatronics' embodies several engineering disciplines in the domains of mechanical systems, electronic systems, control systems, and computer systems. In such a system, the major portion of system functionality is realised by implementing control systems via computer software and hardware, which is mainly addressed by CSDE in VIR-ENG (CDE covers issues closely related to hardware such as developing sensors).

Since the reference model of the VIR-ENG integrated environments encapsulates the roles of the environment, the roles of CSDE can be identified as shown in Figure 3.6. Following the "Control perspective" axis, it is evident that CSDE belongs to the control environment. While DRE covers machine operation, CSDE is strongly associated with the machine design following the "Design perspective" axis. From the "simulation perspective", CSDE supports both the "Real" and "Virtual" aspects.

Therefore, CSDE can be represented by the two cubes at the top corner of Figure 3.6. In summary, the main roles of CSDE are designing the control system within the "Real" and "Virtual" world.

CSDE should facilitate concurrent control systems design processes co-existing in real and virtual environments. The control logic developed within the virtual environment can be transferred to the real environment to control the associated machines / automation systems. Conversely, the control logic used in the real environment can also be transferred to the virtual environment to analyse system performance. There should be little modification required to translate the control logic. The main modifications would be re-mapping virtual components to real components, or vice versa. The control architecture will guarantee the accomplishment of this re-mapping. Consequently, it enables control systems to be developed and tested before the real machine systems are available; it also aids in the analysis and redesign of control systems through the use of virtual machine systems during their operation phase.

However, the machine system design process is a complex multidisciplinary design problem. The complexity arises from the multidimensional interdependency across the involved domain technologies. Therefore, consideration of the interactions with the other environments in VIR-ENG is essential to develop CSDE. The interactions between CSDE and the other environments can be identified through this reference model, which are represented by arrows in figure 3.6. In the control domain, the interaction between CSDE and DRE can be summarised as: (i) for both the "real" and "virtual" world, CSDE interacting with DRE should generate component specifications to guide DRE design and on-line updating of components and facilitate DRE requirements to generate component specifications; (ii) in the "virtual" world, CSDE should facilitate DRE to verify the support system implementation before the machine is ready. In the mechanical domain, the interaction between CSDE and MMDE can be summarised as: (i) in the "virtual" world, CSDE can take advantage of the virtual models generated by MMDE to carry out more accurate simulation; (ii) in the "real" world, CSDE and MMDE work closely to realise the control system and the mechanical system of a final machine. The detail of these interactions is described in chapter 6.

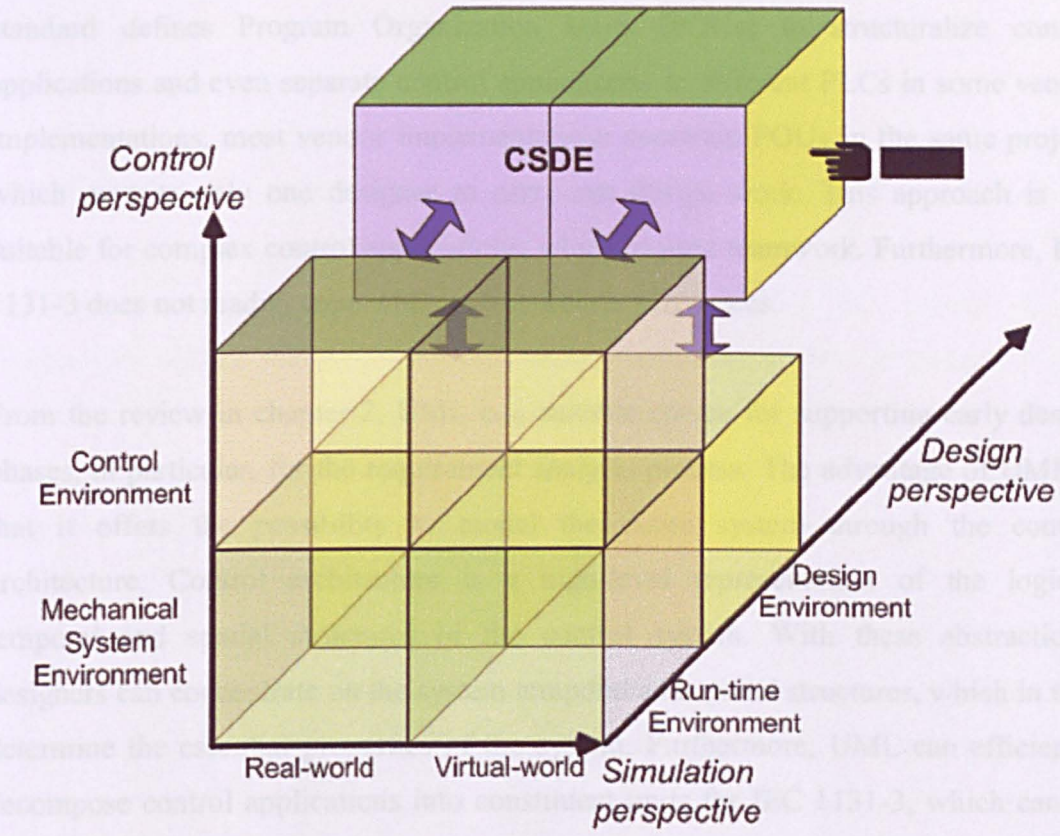

Figure 3.6 CSDE in the Reference Model

3.3.2 CSDE Conceptual Solutions

According to the roles of CSDE in VIR-ENG, it is necessary to choose a programming language to facilitate control system design. Many machine control applications in manufacturing involve predominantly sequential logic (e.g. assembly machines and transfer lines) (Harrison *et al.* 2000). Since IEC 1131-3 languages are well suited to implement sequential control applications and allow users to develop complex control applications without steep learning curves, it is a main player to be used in CSDE. In addition, mechanical engineers are familiar with IEC 1131-3 languages, in particular SFC, to describe machine event specifications. However, current practice mainly employs low level programming languages, such as ladder logic and assembly code in instructions, and does not support efficient program modularization or symbolic representations of I/O and memory variables (Bonfè and Fantuzzi, 2000). For historical reasons, the approaches adopted by PLC design practice normally treat control applications as a single control unit. Although the IEC

standard defines Program Organization Units (POUs) to structuralize control applications and even separate control applications to different PLCs in some vendor implementations, most vendor implementations constrain POUs in the same project, which permits only one designer to carry out design work. This approach is not suitable for complex control applications, which require teamwork. Furthermore, IEC 1131-3 does not readily cope with control architecture issues.

From the review in chapter 2, UML is a suitable choice for supporting early design phases, in particular, for the requirement analysis process. The advantage of UML is that it offers the possibility to model the entire system through the control architecture. Control architecture is a high-level representation of the logical, temporal and spatial structures of the control system. With these abstractions, designers can concentrate on the system temporal and spatial structures, which in turn determine the essential properties of the system. Furthermore, UML can efficiently decompose control applications into constituent units for IEC 1131-3, which can be simultaneously developed. Furthermore, it facilitates the control system requirements to be mapped to the system decompositions and remedy any inconsistencies as the design processes.

The component-based paradigm is the underlying essence in guiding the control architecture design as well as the control system implementation. The basic concept is that the control system can be built by software and hardware components in different hierarchical layers. The constituent parts in these layers have multiple interrelations established by the adopted control logic, the chosen software implementation and the shared hardware resources. The component-based approach maintains the interrelation constancy by defining component interfaces, so that these multiple interrelations can be maintained properly. Consequently, control components can be simultaneously built and the control system can be realized by aggregating control components. Furthermore, control components can be replaced or upgraded, so that it also facilitates required control system reconfiguration.

Based on the proposed solution, it could be appropriate to further divide the entire control system environment into two parts. One part corresponds to control architecture design and the other part covers control system implementation. The two

elements are termed Control Architecture Design Environment (CADE) and Control Logic Programming Environment (CLPE) as shown in figure 3.7.

CADE consists of a defined methodology and associated supporting tools. In terms of methodology, 'Design by Reference' and 'Component Responsibility and Collaboration' (CRC) processes are utilised for realising the control system architecture design. UML is selected as the main description language. It also provides tools to facilitate the creating, housing and categorising the design patterns. The resulting control architecture design is used as the blueprint the forcomplete control system. The detail of CADE is addressed in chapter 4.

CLPE is built utilising the ISaGRAF environment that provides the basic platform for IEC 1131-3 languages. It provides the methods and tools to integrate various control components and systems (e.g. SDS and LON control networks and devices). It also provides tools to repackage existing developed control programs to become components. The approach enables the realisation of component-based control system development via the IEC 1131-3 languages. The detail of CLPE is described in chapter 5.

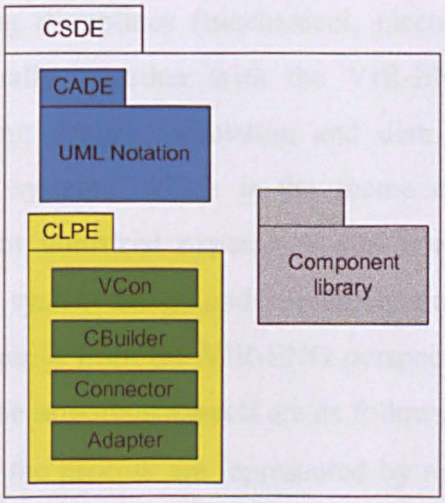

Figure 3.7 CSDE Workbench Structure

One additional part is a component library supported by IIS. During the conceptual control system design, CADE will use the component library to identify existing

components in order to promote reuse of proven systems and design speed. CLPE uses the component library as a component source to form the complete control system and as storage for capturing the design and components for future reuse.

3.4 The VIR-ENG Machine System Design Process

The VIR-ENG design process has been formulated as guidelines for efficient use of the VIR-ENG integrated environment for building machine systems. It was devised specifically to provide a roadmap to guide VIR-ENG users as to design and support, which environments could be used through the machine system lifecycle. It is assumed that machine design and build rely on the experience and expertise of the design engineer and involves various iterative processes, where strict compliance may be either inappropriate or impossible in many cases. Conversely, the guidelines are considered as a reference for the use of different VIR-ENG tools in supporting the design of a machine system.

The VIR-ENG machine system design process is defined as a sequence of iterative activities carried out by a co-operative group of machine designers and engineers from several engineering disciplines (mechanical, electrical, electronic, computer science, etc.). Conceptually, together with the VIR-ENG tool-set, it forms the framework for integrating design, simulation and distributed control of modular manufacturing machine systems, which is the theme of the VIR-ENG research project. Although current industrial experience and practice has been taken into account, such a machine system design and implementation process is introduced by the VIR-ENG team primarily from the VIR-ENG perspective. The design process is shown in Figure 3.8. Some important aspects are as follows:

- The activities within the process are represented by rectangular boxes. Coloured rectangular boxes denote the activities directly supported by VIR-ENG environments. The colour of the boxes indicates ownership of the specific environments. Some blended colours reflect integration between environments.
- The process identifies iteration loops to successively refine designs at several stages. There are four iteration loops (or spirals) involving: 1) conceptual design,

2) mechanical design, 3) control logic programming and 4) run-time support implementation.

- The previous stages in the design process can be revisited if problems or errors are encountered in later stages. In some cases, the starting point may not be the beginning, when retrofitting or re-configuring an existing machine system.
- An information flow is implicitly presented in the process flow from user requirements through to the machine systems being commissioned and used.

Although the design process is similar to classic lifecycle models such as the waterfall model, V model, and X model (Royce, 1970; Hodgson, 1991), where actions are sequentially carried out, the difference can be classified as the anticipation of th designers involved. In a classic pure waterfall design process, the design and analysis must be completed at a certain stage. For instance, in machine design, electrical engineers and control engineers usually start their work after the machine design or mechanical aspects design has been largely completed. In contrast, the VIR-ENG design process emphasises early involvement of the various design disciplines by means of sharing certain discipline knowledge before the design activities begin. Such practice has many advantages, for example, electrical and control engineers could envisage potential problems at an early stage of mechanical design. On the other hand, the development of a machine system cannot be achieved by only involving one or two areas; it requires systematic development covering all aspects. Furthermore, in a classic design process, the mechanical design team usually does not pay much attention to the final control system provided by the control engineers. In the VIR-ENG design process, control system design and mechanical system design refer to a common control architecture specification and control design is verified by using simulation. Obviously, personnel from management, sales and marketing and the client also play vital roles in the process. They do not interact directly with the VIR-ENG environments, but contribute as a 'spectator reviewer' of the outputs at various stages.

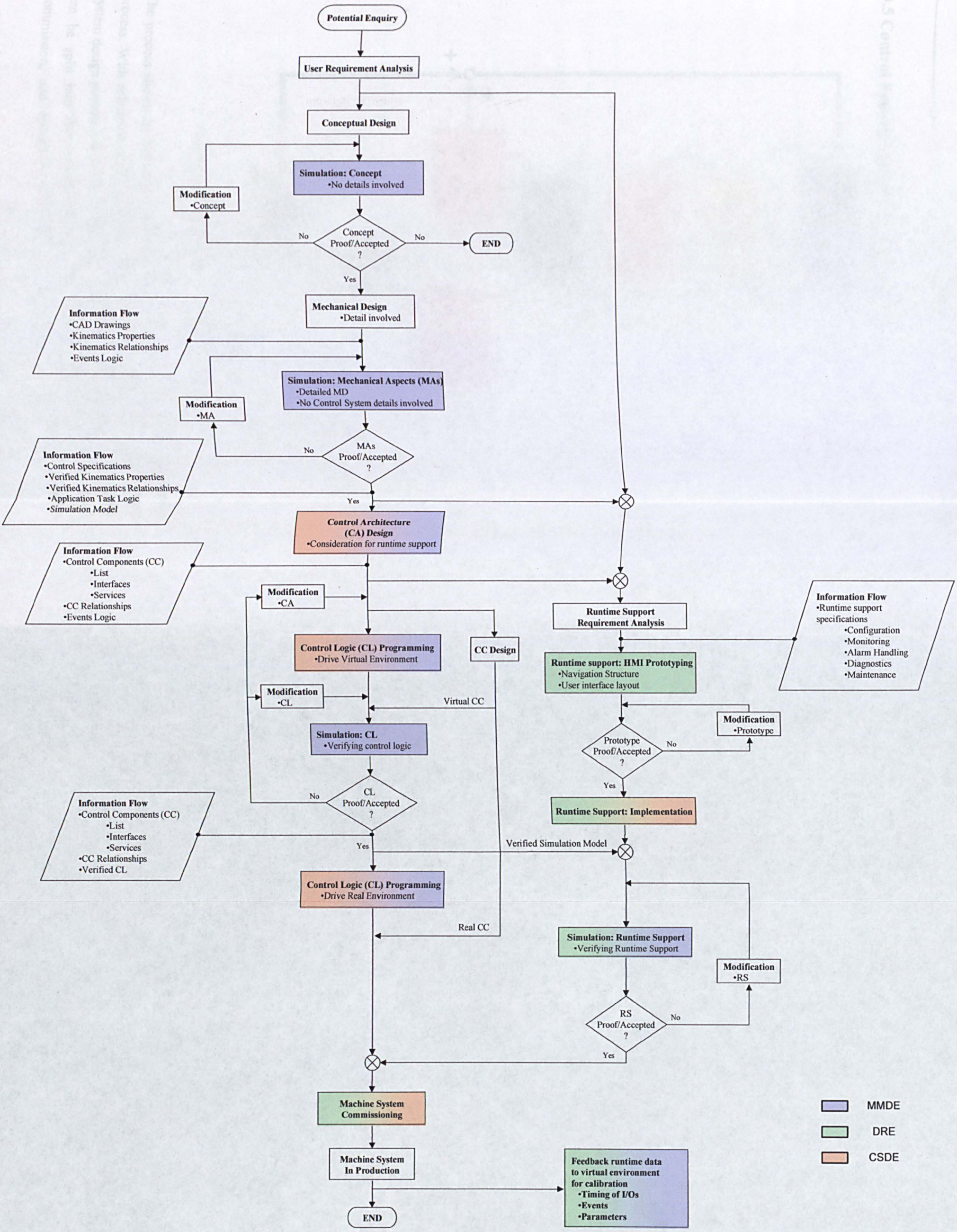

Figure 3.8 VIR-ENG machine system development process

3.5 Control System Design Process via CSDE

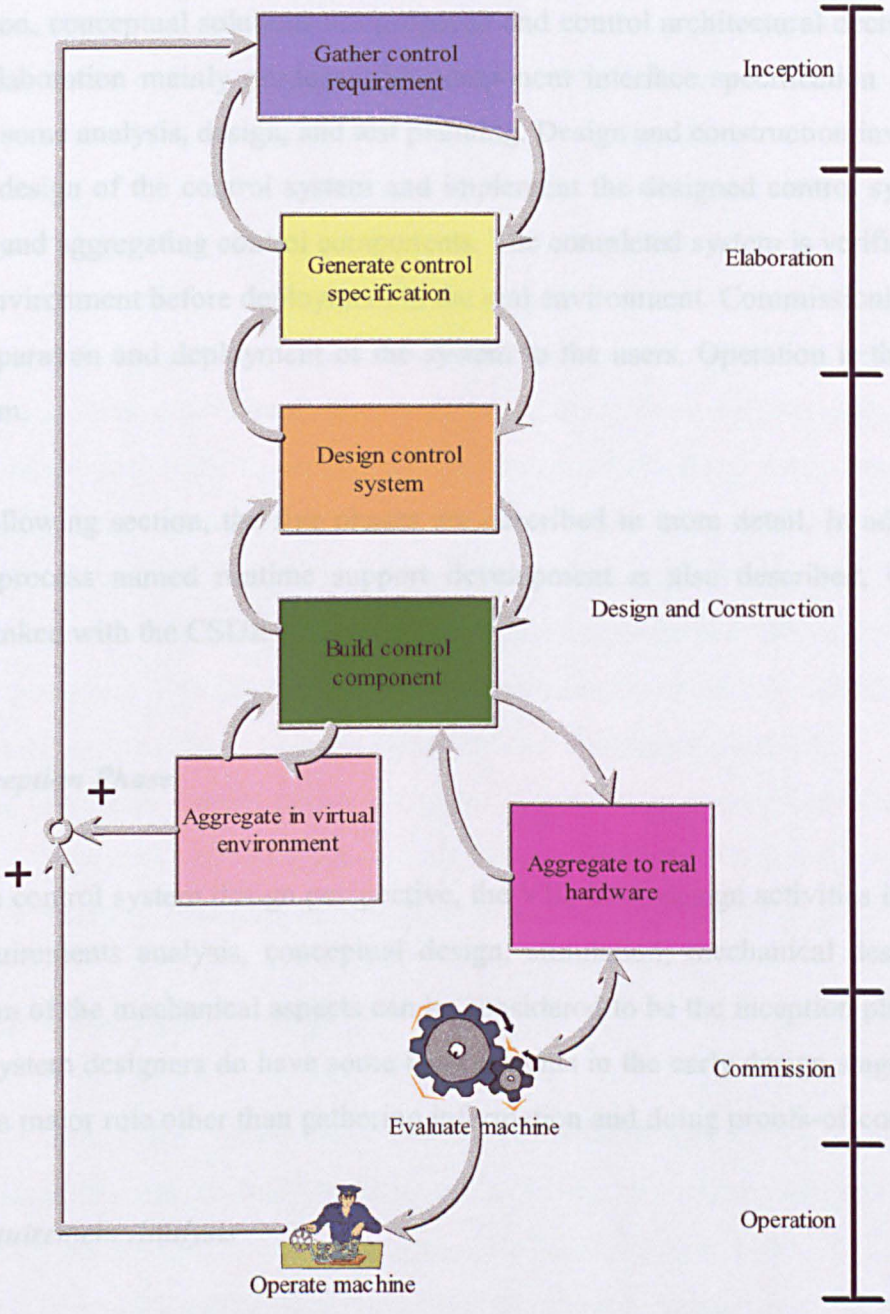

Figure 3.9 Control System Design Process

The process shown in figure 3.8 describes the entire machine system development process. With reference to the VIR-ENG design process, the phases in the control system design process via CSDE can be depicted as in Figure 3.9. The design process can be split into five phases: inception, elaboration, design and construction, commission, and operation. Inception is the beginning of the design process,

involving gathering requirements and doing proof of concept outline. At the end of inception there is a formalised control system requirements specification. In elaboration, conceptual solutions are proposed and control architectural decisions are made. Elaboration mainly produces the component interface specification and also includes some analysis, design, and test planning. Design and construction involve the detailed design of the control system and implement the designed control system by building and aggregating control components. The completed system is verified in the virtual environment before deployment in the real environment. Commissioning is the final preparation and deployment of the system to the users. Operation is the use of the system.

In the following section, the five phases are described in more detail. In addition, a parallel process named runtime support development is also described, which is closely linked with the CSDE.

3.5.1 Inception Phase

From the control system design perspective, the VIR-ENG design activities including user requirements analysis, conceptual design, simulation, mechanical design, and simulation of the mechanical aspects can be considered to be the inception phase. The control system designers do have some involvements in the early design stage, but do not play a major role other than gathering information and doing proofs-of-concept.

User Requirement Analysis

A machine or machine system design life cycle typically starts with the user requirement analysis stage in order to acquire a requirements specification. This stage usually involves discussions between the customer (or potential customer) and personnel from the sales and marketing department. As the result, a requirement engineering process is performed to formulate a machine system specification, which represents a 'wish list' of the machine or machine systems. In fact, a part of the control system requirement comes from this 'wish list'. The 'wish list' involves both functional (e.g. response time for safety systems) and non-functional aspects (e.g.

specific hardware). This is an iterative process among the processes of user requirements understanding, requirements analysis and machine or machine system specifications approval. There is no VIR-ENG tool directly supporting this stage even though the involved personnel may use the simulation tools as a communication aid during discussions with the customer.

Conceptual Design

The conceptual design is to formulate conceptual ideas to meet the requirement specifications. The requirement specifications resulting from the previous stage are both the inputs and goals to be met of the conceptual design process. The process generates solutions without detailed design parameters, and acts as a blueprint for the subsequent design and implementation stages, if accepted by the customer. Conceptual design mainly involves mechanical designers and the corresponding technical personnel. The 3D graphical virtual environment offered by MMDE can be employed to facilitate the communication between different parties (e.g. designer, sales and marketing personnel) and convey the solution concepts to the parties involved. The control system designers can give some advice such as the layout of the machine or machine system, but CSDE tools are not involved at this stage.

Simulation of the Solution Concepts

In order to verify the conceptual solutions, the MMDE tool set is used to build simulation models and conduct simulations. The Event logic of the conceptual models is described using IEC1131-3 languages. It should be noted that at this stage IEC1131-3 languages are considered as logic description languages rather than PLC programming languages. Sequential Function Chart (SFC), a simplified Petri-Net notation, is employed as the main language to describe the high-level sequence and co-ordination between the conceptual entities.

Although the models constructed are still abstract and incomplete, the event logic and simulation models developed at this stage are executable. This allows exploration and testing alternatives and finding problems in the early stage of machine design. Therefore, simulation is not only employed as a concept verification tool but also its

findings are useful as the feedback to further improve the conceptual solutions. The event logic is a vital input for control architecture design.

Process engineers and simulation engineers play an important role at this stage. Participation of electrical engineers and control engineers may start in this early stage to help in such aspects. The co-operation of the machine design team and control team helps the electrical engineers and control engineers to have a better understanding of the proposed machine. In return, it also helps machine designers to consider factors related to the control system.

Mechanical Design

Similar to traditional machine system development, it involves the design of all the mechanical aspects in full detail to meet the machine or machine system requirement specifications. Because sophisticated CAD/CAE tools are widely available and utilised in industry, VIR-ENG environments import models from CAD/CAE rather than provide any tool to directly support the mechanical design. However, VIR-ENG does provide tools to organise the imported models to establish a mechanical component library. Consequently, rather than building systems from scratch, designers could rapidly realise a mechanical system by means of assembling existing mechanical components from the mechanical component library. During the mechanical design process, the control engineer may give advice related to control system issues such as the positioning of sensors. The following items are the major outputs from the mechanical design process.

- CAD models of individual machine components
- Machine layout also represented in the form of CAD models.
- Kinematics properties (e.g. maximum allowable velocity, *etc.*)
- Kinematics relationships (e.g. 'soft' gearing ratio, *etc.*)
- Events logic (e.g. sequence, timing and synchronisation)

The outputs produced at this stage will serve as the essential inputs for building the more concrete simulation model in the next phase.

Simulation of the Mechanical Aspects

The objective of this phase is to evaluate, compare and verify the mechanical design. From the control system perspective, this step is to generate a concrete control system requirement for the subsequent control system design process. The mechanical CAD models that are developed in the mechanical design stage are converted to the 3D graphical objects, which represent the virtual machines/components in the simulation. By utilising the kinematics modelling functions provided by the simulation software, the kinematics behaviour of the virtual machines/components can be visualised based on the designed kinematics properties and relationships. For the control system, kinematics coupling of mechanical parts plays a vital role for control component design and implementation. The event logic drives the models within the virtual environment, so that the mechanical design can be verified. In addition, it also describes the machine system operation processes, which are required to be achieved by the control system.

Simulation provides collision detection for verification, which helps a designer to find out possible collisions due to either incorrect event logic (soft fault) or faulty mechanical design (hard fault). In addition, the collision detection gives some essential information about the machine operation limit for the control system. Simulation also helps to verify the performance of the design, so that simulation outputs (e.g. product flow rate) can be compared to the desired outputs. As a result, any discrepancy between the requirement specification and the design can be found. Through some translation, the simulation output can also be converted to some vital information for control system design, such as the desired response time.

3.5.2 Elaboration Phase

Through the inception phase, a concrete control system requirement is available for control system design, which combines the requirements directly from the customer and from early design activities. The elaboration phase of the design process includes some planning, analysis, and control architecture design. Elaboration includes several aspects of a control system development, such as presenting proofs-of-concept,

developing test cases, and making design decisions. The elaboration phase focuses on setting the architectural foundation for the control system. It involves the important engineering activities that transform the control requirement specifications into control architecture, which is to describe the blueprint of the control system structure.

The control requirements are gathered into a document called a control requirement specification. They consist of two major items: 1) verified kinematics properties and kinematics relationships between machine components; 2) application tasks logic described in IEC 1131-3 languages. In the process, MMDE tools are helpful in terms of capturing and visualising the control system requirement. The executable virtual model developed at the previous stage provides a way for control engineers to intuitively understand the operation process better than to simply reading the document, as in the traditional design process. Furthermore, after the control architecture design, the control engineer can use the virtual model to verify the design by using specific scenarios, so that design flaws are easily identified.

Other tasks in elaboration include refining the initial estimates, reviewing the control requirement specification, and investigating risks. CADE can help in formalising and analysis of requirements and facilitating control architecture development.

The following items are the major outputs from this control architecture design process:

- Control component list
- Control component interfaces
- Control component responsibilities
- Relationships between control components
- Required operational sequences
- Exception handling strategy

The elaboration phase is over when the high-risk and architecturally significant control components have been fully detailed, proofs-of-concept have been completed, and the potential exceptions have been identified. In other words, this phase is complete when the control architecture has been finalised.

3.5.3 Design and Construction Phase

During the design and construction phase, the remainder of the control system is analysed, designed and built. Using the control architecture from the elaboration phase as a foundation, the control engineering team will build the remainder of the system. In detail, the tasks in the design and construction phase include control component design, control logic programming, simulation in the virtual environment, and assembly in the real environment.

The control architecture can be considered as a control system specification to be implemented. However, this control system specification focuses on conceptual design; it needs to be refined and further developed. Consequently, the detailed control system design refines the control system specification by committing to enabling technologies.

First step for the detailed design, is to divide the components based on dependence of hardware (referring to the four-layer model). The control system specification produced in the control architecture design consists of all four layers. While Device Components and/or Composite Components are considered as hardware dependent components, composite components, modular machines and machine systems are hardware independent components. The composite component is a grey area, which is largely determined by the system structure.

Within VIR-ENG, CSDE is not supposed to deal directly with hardware dependent components, which are covered by the Component Design Environment (CDE). CSDE will pass the component specification to CDE, which is intended for identifying off-the-shelf components or for developing new components as required. The component developed in CDE needs to be compliant to the VECOM model. Subsequently, CSDE acquires the defined component from CDE and imports it into the Control Logic Programming Environment (CLPE). Meanwhile, CDE also provides the associated virtual components for simulation purposes.

For hardware independent components, CLPE provides the workbench and associated tools for coding via IEC 1131-3 languages, by means of composing the existing components and generating the new components based on the component interface specification. For this research, SFC is chosen and used to ground the control architecture design result in a workable form by specifying the required resources (e.g. memory sizes) and system platform (e.g. SDS or LONWORKs).

The programming (or coding) is carried out in the IEC 1131-3 programming environment. The design engineer can choose between developing the control logic components from scratch or using the existing control logic components and modifying them according to the control system specification.

After coding, the prototype system will be simulated in the virtual environment. During the simulation process, it facilitates evaluation of performance. Data gathered during the simulation (e.g. cycle times for individual workstations and throughput of the system) are used for analysis. If the design fails to fulfil the performance specifications, either the control logic or the control architecture has to be modified. The worst scenario involves changing the mechanical design.

Once the prototype fulfils the requirements and the physical facilities are available or partly available, the prototype is further developed and finally aggregated into the real machine. The verified control logic is used to drive the real devices/machines. From the VIR-ENG perspective, it involves connecting the control logic programs to real control components. As a matter of fact, the simulation is an ideal environment. As result, some information is associated with the physical facilities such as hardware specific properties, error detection, and some further control logic needs to be developed. On the other hand, it implies that control engineers, developing the control system, do not need to be concerned with the particular hardware in the early stages.

Compared to the traditional design process, developing the control system in the virtual world has a big advantage in terms of reducing development time and evaluating the design. In the virtual world, enhanced graphical representation and visualisation facilitate the discovery of potential logic errors (e.g. collisions between

parts). The majority of the control logic can be fully tested before it drives real machines, including some exception handling.

Design and construction is over when the control system is complete and tested. It is important to ensure that the control architecture and control system are synchronised; the control architecture will be extremely valuable once the control system enters the maintenance and reconfiguration mode.

3.5.4 Runtime Support Development

As figure 3.9 shows, there is a parallel sequence of design activities in the Distributed Runtime support Environment (DRE). DRE aims to provide configuration, monitoring, alarm handling, maintenance and diagnostics to the operator. Main design activities involve runtime support requirement analysis, human machine interface prototyping, implementation of the runtime support system and verifying the runtime support system using simulation.

In the analysis phase, DRE analysis and design can be carried out in CADE during the control architecture design process. However, there are some differences between DRE and control system design. CADE facilitates DRE design activities by means of adding in extract notations.

During the HMI prototyping process, the navigation structure of the runtime support system will be used to depict (with its navigational class) a customised view of the application tasks, which are also facilitated by CADE. Once the navigation structure is established, visual elements (or components) can be defined. The navigation structure serves the following purposes:

- Presents a customised view with regard to user types and their associated tasks;
- Easily locate the required information;
- Prevents the users from being disoriented while browsing through the system information.

Carrying out DRE design within CADE gives a big advantage in that it closely links the supporting system design with the control architecture. Modification of the control architecture can rapidly affect the supporting system design; the change of the supporting system design can also be reflected in the control architecture.

3.5.5 Commissioning Phase

Similar to a traditional machine development cycle, the machine system-commissioning phase in the VIR-ENG process involves testing and evaluating the machine or machine systems on-site. The control components developed using CSDE will now serve as the controllers of the real devices. At the same time, the RTS developed using DRE will be used as the monitoring and testing front-end for the shop floor personnel when testing is conducted on the machine system.

3.5.6 Operation Phase

In traditional machine design practice, this is the end of the life cycle. The control components developed using CSDE serve as the controllers of the real production machine. At the same time, the RTS developed using DRE is used as the monitoring and diagnostic front-end for the operator/shop floor technicians in their daily operations. Since the completed machine system is handed over to the user community, the design tools will disappear. After the designed system is implemented, there is limited, if any, future need for the control architecture results and the virtual models, and they are abandoned. As a result, many virtual models are never verified because most models support the design of a proposed system.

However, it is not the case in this design process; the design tools will be kept active during the operation phase. During normal operational periods, run-time data will be gathered from the production process and fed back to the virtual environment for model calibration. The enhanced model can subsequently be employed for tasks such as production planning and scheduling, operator training, process optimisation etc. When unpredicted changes occur, the combination of the requirements and the virtual

model associated with the real machine can be used to carry out another design cycle and finally reconfigure the system.

3.6 Summary

In this chapter, the key features of VIR-ENG methodology can be summarised as:

- Three perspectives have been identified, which are Design, Simulation and Control to form a three dimension functional space. Following the perspectives, the entire machine system design roles have been systematically identified.
- A component-based development paradigm has been adapted for the overall strategy.
- The integration between the virtual and real environments has been emphasised to facilitate the machine design process.
- The support for the complete machine system life cycle has been stressed to facilitate system design as well as system maintenance and reconfiguration.
- A workflow has been proposed to support an iterative design process for machine system design.

CSDE derived from VIR-ENG facilitates the machine control system design process. As the basic principle, CSDE uses the control architecture to drive the component-based control system design. Consequently, CSDE is further divided into two essential elements, namely, CADE and CLPE. CADE uses UML to implement conceptual design of control systems and CLPE adopts IEC 1131-3 languages to design and construct the control systems.

A design process for the CSDE is proposed as part of an overall VIR-ENG machine design process. One of the major benefits from integration within the VIR-ENG environments is to design and construct the control systems within the virtual environment.

Chapter 4 Control System Architecture Design

4.1 Introduction

For machine control system development, control system architecture design is the first step that begins to place requirements within a control solution space. The architecture determines the structure and management of the development project as well as the resulting system, since teams are formed and resources allocated around architectural components. For complex control systems, the overall system structure or control system architecture emerges as a critical design problem. The importance of control architecture for practicing engineers is confirmed by estimates indicating that 60-75% of life cycle costs are committed by the completion of architecture design. From a control system perspective, architecture means the structure or structures of the control system, which comprises software and hardware components, the externally visible properties of these components and the relationships among them.

For component-based machine control systems, the key elements are the components that include hardware components and software components; the entire control system can be constructed from a set of specific components. Software components do not comprise the entire control system; in fact, hardware components establish the foundation of software components in the control system (Gupta and Buzacott 1989; Pyoun and Choi 1994). Software components play an important role in a machine system, which need to integrate every part of the machine control system together into an orchestrated flexible robust system. The proposed control architecture design method for component-based machine control systems takes advantage of the development of object-oriented or component-oriented analysis and design methods, which originate from the computer science community. In particular, the method adopts the use of different levels of abstraction and different views, which is advocated by popular methods. Design issues at this stage include gross organization and control structure, assignment of responsibilities to components, and interaction between components.

This chapter outlines the proposed method named Component Responsibility Collaboration (CRC) and the associated graphical aided tool, namely Control Architecture Design Environment (CADE). It starts from a discussion of the control system requirements. As the design methodology, CRC approach adopts the UML notation as a description language to document design results. In detail, CRC employs four diagram types: task decomposition diagram (a normal perception of the system), deployment view (overall component relationship), logic view (a detailed description of the component interface), and process view (a component dynamic relationship). The proposed design sequence of the CRC method is also presented. Associated with the CRC method, CADE provides an interactive graphical environment to facilitate control system architecture design via graphical and textual languages.

4.2 Control System Requirements

With reference to the overall VIR-ENG workflow, the control architecture design starts after the beginning of mechanical design, which is mainly effected by MMDE. MMDE mainly deals with mechanical design, but in some senses it determines the main quality and function of the final machine system, such as standard operation process and rough production. Thus, control system requirements come from two parts, which are the original one directly from the customer and another one from MMDE. Assuming the mechanical design is at least partially completed, there are several important information items that come from MMDE:

- The functional requirement of the control system. It describes the function that needs to be achieved by the control system, such as default velocity and axis movement.

- Mechanical specification, including the capability of the mechanical system such as minimum and maximum velocity, kinematics properties, realised actions by the mechanical system and responsible exiting (or legacy) components.

- Quality requirements, which need to be implemented in the control system, such as movement accuracy and response time.

- Machine operation sequence or event logic, including normal operation sequence, system shutdown sequence and system-resuming sequence.

In addition, the structure of mechanical modular entities (components) is necessary input for the control architecture design, which reflects design decisions to cope with unpredictable changes in mechanical aspects. Subsequently, control components design needs to be incorporated with the mechanical design decisions by means of aligning the control component structure with the designed mechanical structure, so that the different perceptions and experiences are unified. As a result, the machine system can react to changes as a whole. In this sense, control component specification needs to be verified from a mechanical design perspective.

4.3 CRC (Component Responsibility & Collaboration) Approach

Previously, several methods for the analysis and design of component-based systems have been published (Selic *et al.*, 1994; Malan *et al.*, 1996; D'Souza and Wills, 1999). However, they are mainly formulated from a software perspective and do not completely meet the requirements of designing and constructing machine control systems. In many cases, these methods offer specification techniques that are redundant or require too much detail, or they are too rigorous in their procedures. This is caused by the fact that most of the required functionality and hardware components are already captured. The objective of control system development is the design of the software control process of hardware components.

The CRC methodology intends to address this issue. The CRC method can be used to determine the architecture of a control system, which can also be viewed as conceptual solutions in fulfilling the requirements as specified for the control system. Although the CRC methodology proposed uses the same acronym as the CRC card, and part of the concept also inspired by it, subtle differences do exist between them.

- The first 'C' represents 'component', not 'class'. This 'C' covers 'software' and 'hardware' components, so that it expresses the fact that a control system involves two types of component.

- Initially, this approach is focused on finding responsibilities rather than components.

- 'Re-use' is emphasized both for 'hardware' and 'software' components, rather than only for 'software' components. In practice, they are equally important in control system design.

- The outcome of a control system solution is the component interface specification.

- The CRC card does not suggest any structure to organize the 'class', but this approach does suggest a hierarchical tree-like structure to organize the 'components'.

As outlined above, the input to the CRC method is a list of control system requirements. These requirements include functional, quality, and constraint aspects. The requirements elicitation and analysis activities do not have to be completed prior to beginning the design. The beginning of design activities does not mean that requirement elicitation and analysis are stopped, they will be continued alongside the design activities.

The CRC method is an integrated design method that supports a blending of classical top-down and bottom-up approaches. Most methods of Object-Oriented Analysis and Design (OOAD) for software systems focus on the definitions of individual objects and their relations (Meilir, 2000). Conventionally, this approach is recognised as an effective way to support bottom-up fashion rather than top-down, since OOAD does not formalise and elaborate object decomposition. In the bottom-up approach, characteristic of an engineering design practice, design is built from known components in anticipation of fulfilling functional requirements. While OOAD is increasingly adopted by industry for large-scale systems development (Balakrishnan

and Somasundaram, 2001), the problem quickly leads to combinatorial explosion, inefficiency in the design process, and difficulties assessing the effectiveness and merits of design alternatives. On the other hand, in the top-down approach, characteristic of a system engineering process, design is driven by functional requirements toward solution alternatives. While designed solutions using this approach seem to meet functional requirements, there is no guarantee that solutions are realisable in terms of physical manifestations. In sum, the top-down or constructive way is the most effective way to tackle the complexity of control systems; the bottom-up or evolutionary way is extremely useful for re-using off-the-shelf components. Taking advantage of both approaches, the CRC method provides the frequent iteration between abstraction associated with control requirements and detail associated with components. The design of control architecture is based on the process to decompose global specifications and requirements into responsibilities, together with the process to compose hardware and/or software components to achieve responsibilities. This merged approach combines the power of the architecture-based development with the leverage of the building-block approach. This approach is not restricted to a specific design and analysis method or technique in the sense of enforcing those specific rules and method steps. Instead, it provides a general approach with sufficient abstraction to support multiple views switching between design aspects.

4.4 General Development Steps of the CRC

The CRC method for control system architecture design is based on an iterative sequence of steps, which is utilised by four views:

- The task decomposition diagram captures the functional requirements of the control system, and discovers a set of responsibilities. In fact, it establishes a communication bridge between mechanical design and control system design.

- The deployment view describes the overall structure of components, including the relationship between hardware and software components.

- The logic view further describes components by focusing on component interfaces.

- The process view captures the component's dynamic behaviour and relationships.

The different views allow designers to concentrate on the different aspects of the architecture design. The design process integrates four views in a systematic way, which can be depicted as in figure 4.1. Since the task decomposition explicitly handles control requirements, it is recommended as the initial point for architecture design. The design process starts with the decomposition task to formalise control system requirements and discover entire responsibilities of the control system, followed by identifying components to construct control system with hardware and software components associated with proper responsibilities, continues with specifying interfaces to refine the identified components and may introduce some aided components and concludes with the verification of designed components. Through the verification based on the control system requirements scenarios, the design results can be transferred to the latter development. In addition, the components verification integrating with mechanical design results can establish the link between designed components and corresponding mechanical parts. As with all of the steps in the CRC method, executing one step may produce insight that will cause reconsideration of prior steps. Furthermore, discussions that occur during one step may uncover information that pertains to a future step.

The results of the process are summarised into a list of components, each having a set of interface specifications and constraints. After the components verification, the process is supposed to reach the end, but the process could be terminated at any of the other stages. There is feedback between the verification and the task decomposition, but it is also possible for the verification to link with the other activities. The loop helps to ensure that all of the activities have been undertaken. The designer may want to begin with the specifying interfaces, for example, if the system design only involves adding extra functions.

Based on the above description, the whole design process can be broadly divided into two parts, which are discovering responsibilities and component design. In the process to discover responsibilities, the notation of the task decomposition diagram is using boxes and links to represent the idea of the responsibility organisation, similar to an organisation chart. In component design, the Unified Modelling Language (UML) offers appropriate means of basic notation for the rest of the three views to facilitate the component design. UML is a general-purpose visual modelling language that contains a wealth of constructs to fit into nearly every application domain. Besides the basic UML elements, UML introduces extension mechanisms that enable one to customise and extend model elements. Based on these extension mechanisms, extra notations have been adopted to fulfil the features of the control architecture design.

The CRC method proceeds by recursively decomposing the system(s) and constructing the components. The whole process may consist of several design cycles. In practice, it is nearly impossible for the control architecture design to take one cycle only. In addition, it is necessary to consider several alternatives, so that one optimal solution can be found.

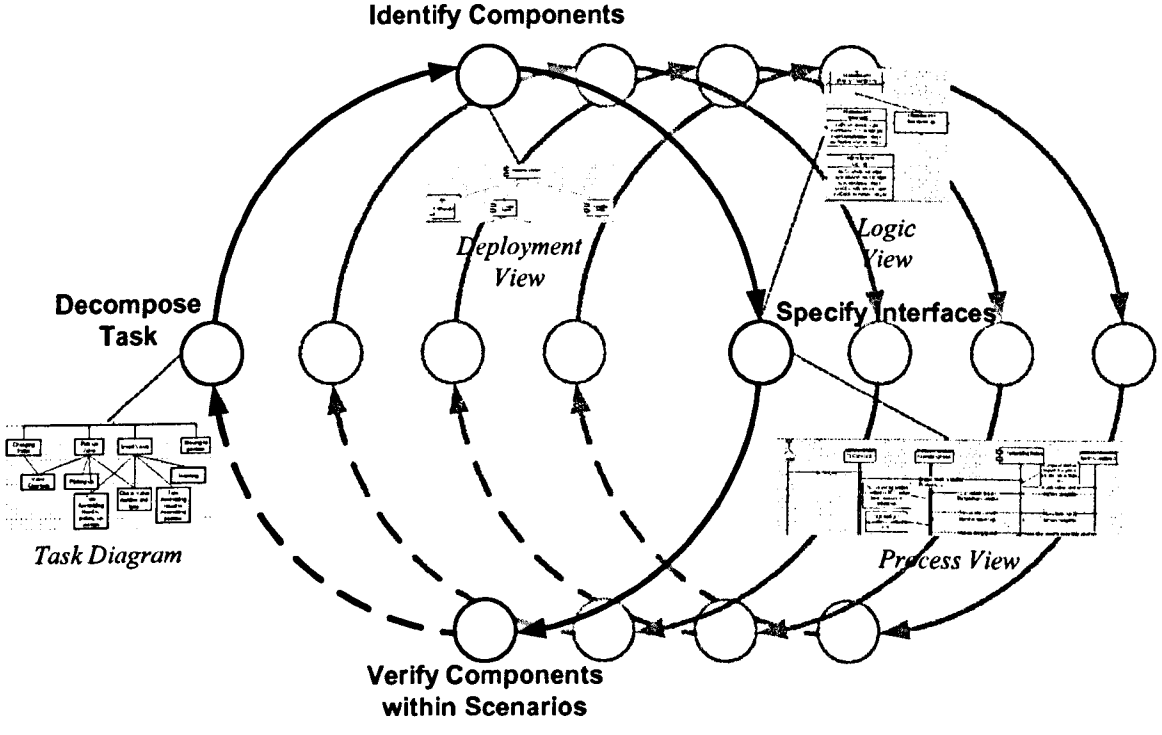

Figure 4.1 General Development Steps

4.4.1 Task Decomposition

Task decomposition is the place to connect mechanical and control design together. In mechanical design, mechanical modules will be assigned to the identified tasks, which present the idea from a mechanical perspective and the strategy for the machine system evolution. After control system design, software and hardware components will be assigned to tasks too. It helps the control and mechanical engineer to understand each other. In addition, it also assists in verifying the whole system for both sides. During a re-design process that includes upgrading and re-configuring process, it allows the engineers to focus on the areas requiring major change.

Task decomposition is a functional approach to knowledge elicitation that involves dividing a problem space into a hierarchical tasks tree, which should be tackled in a subsequent design solution. The aim of task decomposition is identified as determining:

- The objectives of the task
- The procedures used
- Any actions and objects involved
- Time taken to accomplish the task
- Frequency of operations
- Occurrence of errors
- Involvement of subordinate and super-ordinate tasks

The result is a task description, which is formalised in a task tree. It is mainly gained from the knowledge of standard machine operation. The process does not, however, describe knowledge directly. That is, it does not attempt to capture the underlying knowledge structure but tries to represent how the task is performed and what is

needed to achieve its aim. Any conceptual or procedural knowledge and any objects that are obtained come from the control or mechanical engineer's knowledge.

In task analysis, the objective constraints on problem solving are exploited. The method consists of arriving at a classification of the factors involved in problem solving and the identification of the atomic task involved. There is always a single and vital question remaining to be answered: when dose the task decomposition come to an end? As the bottom line for control system analysis, we suggest that the task decomposition is finished when the task decomposition reaches the atomic task. There is no universal agreed definition of an atomic task for control system design. The definition of the atomic task mainly depends on the knowledge of control and mechanical engineers; different people may get to different end points. The general atomic task is that the task can be achieved by using a single actuator or the task relies on the legacy system.

As a means of breaking down the problem area into its constituent sub-problems, task analysis is useful in a similar way to data flow analysis or entity modelling, which is mainly governed from the physical perspective. Although the method does incorporate the analysis of the components associated with each task, it is lacking in graphical techniques for representation of these components. However, it remains mostly useful for functional elicitation.

The method of hierarchical task analysis can be better understood through an example. Figure 4.2 is based on the part of the demonstrator concerned with assembling an engine block. The overall task to be described is that of the assembling operation where the assembly station receives an engine block, inserts valves into the engine block, and sends out the engine block.

Figure 4.2 can be interpreted from the top level, which represents the overall goal of the operation *Assembling Engine Block.* Note that the hierarchy is represented as a net at some points, which is only to simplify the representation and avoid repetition of common subordinate operations and recursion.

The top-level task decomposes into two sub tasks based on the engineering considerations. While *Transferring EB* responses to convey EB in and out, *Assembling Operation* accomplishes the assembling EB. Although these two tasks interact with each other in a serial manner, at some point, they can be carried out simultaneously. For example, during the EB transferring period, the assembling machine can simultaneously carry out the preparation actions for the next assembly operation. Therefore, it is appropriate to separate the whole task into two for consideration.

The next level of decomposition is based on the knowledge of the ordinary operation sequence. For example, the assembly operation cannot be achieved without picking up the valves. Thus, the assembling head will move to pick up position to pick a valve, and then move to insert position to insert it. Although the assembly operation cannot be completed within one time, it does nothing other than repeat the identical tasks. At the end of the process, a set of atomic tasks is reached, which are believed simple and precise enough to be achieved based on the available knowledge and experiences. For example, how to detect the engine block is in the right position is well known. Actually, some atomic tasks could still be considerably complex, because those tasks are utilised by the legacy systems. For instance, the task *changing pallet* uses magazine to achieve it, that machine exists and is ready to run and so we do not need to bother anymore.

In some ways, it could be held that the use of a formal technique such as task analysis in the above example adds nothing, but that common sense could be derived. However, its use in structuring the information derived from engineers' knowledge is invaluable for the following reasons.

- The decomposition of complex tasks into more primitive or unitary actions enables the control engineer and mechanical engineer to arrive at a common understanding of the tasks and the available implementation technology, as has been seen in the above analysis.
- The second factor is that the very process of constructing and critiquing the task hierarchy diagrams helps uncover gaps in the analysis, and thus remove any contradictions.

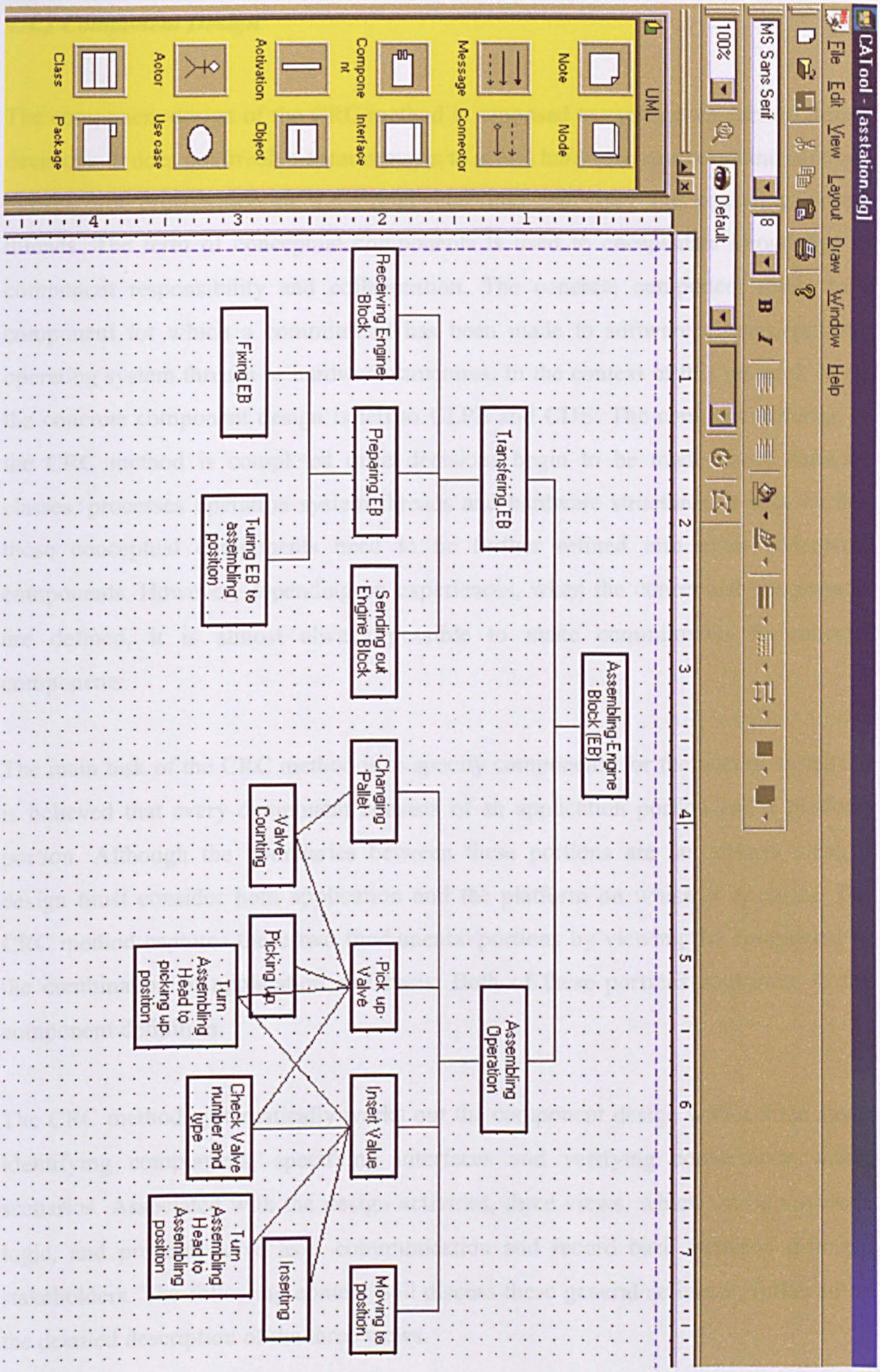

Figure 4.2 The Example of Task Decomposition

4.4.2 Component Design

The component design of the CRC method is supposed to manage the earliest design decisions. It does not involve commitments to actual hardware structures and software classes, nor organisation of the components into processes and operating system threads. The term of conceptual components is used to encapsulate knowledge of component responsibility and collaboration. The concrete component refers to a component for which a commitment has been made to software classes, process, operating system threads or hardware structures. In the context of this research work, the concrete component design is left to CLPE and CDE. The component design of the CRC method is completed once decisions begin to be made about software classes, processes operation system threads and hardware structures. It may be that those conceptual components need to be further refined into extra conceptual components. However, depending on experiences, when the conceptual components are defined, it is almost always possible to make commitments to concrete components.

The main task of the CRC method is to specify components for the overall system. It is believed that every component consists of an application portion and a platform portion. Although the boundaries between these portions are not always clear, a design must consider both application and the platform on which it executes. The CRC method captures these two fundamental portions by viewing the component as the combination of application and basis. Both of these portions contribute to the component definition.

The CRC method systematically works out the component design within three steps: identifying components, specifying interfaces and verifying components within scenarios. Associated with the design activities, three views, which are deployment, logic, and process, serve as a communication and record basis between different stakeholders. The following content will discuss these general activities, followed by the detailed description of the three views.

4.4.2.1 Component Identification

Component identification is the key activity for component-based architecture design. In relation to requirements, it is considered to be a responsibilities assignment process. Initially in the whole process the identified responsibilities are concerned with relating the task decomposition associated with functional requirements and the additional customer requirements associated with non-functional requirements. The objective is to solve potential conflicts between requirements. The next step in the process is to choose a control system architectural style. Based on the review in chapter 2, architectural styles include central, hierarchical, heterarchical, and hybrid. The choice of architectural style relies heavily on the architects' experience in design. The choice of architectural style yields a collection of components. The responsibilities are assigned to the components as they are identified. The identified components can be elaborated either by hierarchical decomposition into sub-components, or by aggregation to a big component. Relatively, decomposition leads to architecture style choice and responsibilities reassignment. It is in the iteration of these steps: component decomposition, choosing styles, and assigning responsibilities that designers must decide whether the compromises that have been made are adequate.

Component identification needs to separate the hardware and software components. The hardware component is made up of its interactions on the factory floor; it achieves the actual execution. The software component is connected to other software or hardware components, which hold information. These two are not necessarily coupled, specially, at the conceptual design stage.

In order to cope with unpredictable changes, the general principle for component identification is to discover generic components, which are broadly recognised by system engineers. Generic components are independent from tasks at hand and consistent solution fragments. The use of generic components leads to constructing reconfigurable (and hence change capable) systems (Weston, 1999). In addition, well-defined components already developed (or under development) could be used in a reuse-intensive way to gain benefit over current best practice.

Based on the features of component identification, it can be mainly carried out in the deployment view providing the system overview. In the deployment view, the presentation can combine hardware and software components into the integrated picture. It also provides the abstract presentation, which focuses on depicting component entities rather than detail. Thus, it can efficiently represent the architectural style of the system, which is the most important part. However, component identification could also occur in the other views. The new components could be found, for example, in the process view, when a mediator service is required to simplify communication implementation.

4.4.2.2 Interface Specification

The objective is to introduce control between components and to refine the configuration, which describes collaboration between components. The interface consists of the data and control flow information needed by each identified component. The formulation of a precise interface specification permits the detailed design, implementation and testing of a component.

The specification can take many forms. One of the most popular is the contract approach. In this approach, the semantics of individual services are specified by the pre-/post-condition pair. Invariant constraints are then used to specify additional semantics on aggregates of these services at the interface.

The interface needs to capture the variations that will occur during component reconfiguring. In some sense, this is the ability to cope with unpredictable changes. The more explicitly these variations can be captured at an early stage, the less problems with design will occur during system development. These variations can be either coarse or fine-grained. For example, an assembly machine may assemble four-cylinder engines or five-cylinder engines or both. The control system must be able to support all three configurations. This is an example of coarse-grained variation. On the other hand, this component may use the LON system for implementation or it may use the NextMove control card for implementation. This is an example of fine-grained variation. The CRC method is concerned with variation at a granularity that has

impact on the conceptual component. This is primarily coarse-grained variation but some aspects of the conceptual component may exist to allow for fine-grained variation.

Commonalties refer to the fixed points within the variation inherent in a component. These may be features that are common to all instances of the component, for instance, if every assembly head needs to assemble five-cylinder engines, the function is in common.

In order to achieve variation, the interface specification may need to introduce new component properties. Through configuring component properties, components could be adapted to different conditions. The redevelopment action should not happen and the interface specification should remain the same.

However, variability can occur either in function, platform or environment. An example of platform variability might be the change of an operating system. Two versions of components may be developed, so that the rest of system remains the same. The CRC method assumes that both kinds of coarse-grained variation (function and platform) are captured during the requirements phase. The mechanism for capturing and representing the commonalties and variability is outside of the scope of the CRC method. Once the variations are captured, the achievement of this variability becomes the responsibility of the component.

This activity is mainly carried out in the logic view and the process view. In the logic view, the component interface is detailed. It also includes the descriptions of the relationship between component interfaces. In the process view, the interactions between components have been described, which leads to the dynamic behaviour and the use of the interfaces.

4.4.2.3 Component Verification

Once component design is completed or partially completed, component verification can determine the appropriate components. It is performed by means of applying

scenarios to the designed components. Scenarios derive from concrete requirements, which stem from MMDE or directly from the customer. The term concrete requirement means that the original requirement has been interpreted and re-defined by control designers. It is not meaningful to have a requirement "control system should be flexible" because all systems are more or less flexible with respect to some sets of easy changes and some sets of difficult changes. A concrete form of this requirement is that "it should be easy to add new features of the following type..." For each scenario, ask whether the component can still satisfy the scenario. Each scenario includes an expected attribute stimulus and a desired response. Consider the decisions made so far in the design and determine whether it is still possible to achieve the scenario. In the event of a negative answer, either the decision that has been made should be reconsidered, or the component must accept the failure to realize one of the scenarios. The rationale for accepting a failure to realize one of the scenarios should be recorded.

4.4.2.4 Deployment View

The deployment diagram is defined in the UML Notation Guide. The purpose of the diagram is to show the configuration of run-time processing elements and the software components that are associated with them. The deployment view employs the deployment diagram to represent overall system components, which facilitates the component identification and documents the results. The notation used in the deployment view is the same as in the deployment diagram. However, some modifications and extensions have been made.

In the deployment diagram, a node represents a run-time computing resource, generally at least having memory and often processing capability, where components can be deployed. Thus, a node represents a piece of hardware. According to the description above, it is believed that the hardware component can be represented by the node. Some hardware components such as normal sensor do not have computing capabilities, but they should also be represented in the diagram. In order to solve this problem, the node used in the deployment view does not necessarily have to have computing capabilities, so long as it is hardware, which can interact with the rest of

the system. A component represents a physical piece of an implementation of a system, including code (source, binary, executable) or equivalent items such as a script or file. Thus, the component is a natural choice to represent the software component.

Although the examples given in the UML Notation Guide only illustrate instances, the node or component could not be represented as a type. It is believed that the node types and component types should be very useful at this stage. A major restriction in UML is that component types are not allowed in deployment diagrams. This prevents the modelling of the overall system structure. Therefore, we need to introduce component type into the deployment diagram to display both hardware component and the relations between hardware component and software component.

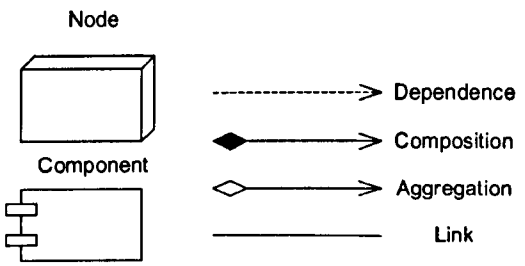

Figure 4.3 Notations for the Deployment View

4.4.2.5 Logical View

In the UML Notation Guide, there are two diagrams related to this issue, which are the class diagram and the component diagram. The class diagram is a collection of static declarative model elements, such as classes, interfaces, and their relationships. The component diagram represents the component connection by dependence relationships. Since the logical view is used to record the responsibilities and conceptual interfaces for the component, the class diagram is more suitable to present the logical view. Furthermore, because the logical view should not commit to specific codes or files, which is one purpose of the component diagram, the component diagram is not very helpful in this case. The logical view uses the UML standard

interface notation to describe the component interface and includes the connection notation to describe the relationship of component interface.

As for the deployment view, both software and hardware components should be considered in this view. Because the control system development is considered as a software intensive process, this view is more focused on software component interface design. For a hardware component, its interface is to describe the programmable interface rather than design interface. It is useful at the system installation stage, which finally integrates all of components together to form a complete system.

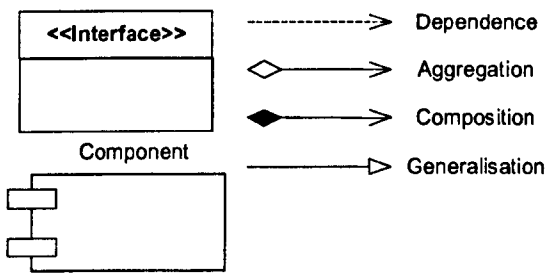

Figure 4.4 Notations for the Logical View

4.4.2.6 Process View

The process view represents the dynamic behaviour of a component interface. In the UML Notation Guide, two diagrams could be used for this purpose. They are the sequence diagram and the state-chart diagram. A sequence diagram presents an interaction by a set of messages to effect a desired operation or result. A state-chart diagram represents the dynamic behaviour of entities by specifying their response to the receipt of event instances. Since the information used to represent the dynamic behaviour of the components comes mainly from the event logic that is represented by SFC, the sequence diagram can easily transfer these descriptions. Another fact is that the sequence diagram can clearly display the component involved. Therefore, the sequence diagram has been employed to represent the process view. However, in the sequence diagram, it uses the object or object type (not component type) is used to

construct the diagram. From the process description point view, the component type can be represented by using object type, since component type has been assigned responsibilities, which are owned by the object.

The process view helps to refine the component interface specification defined in the logical view. On the other hand, the designer can express his ideas about how the component interfaces to fulfil the operation in the requirement. But early commitment should be avoided, e.g. specific operating system calls, hardware dependent sequences, should not be represented.

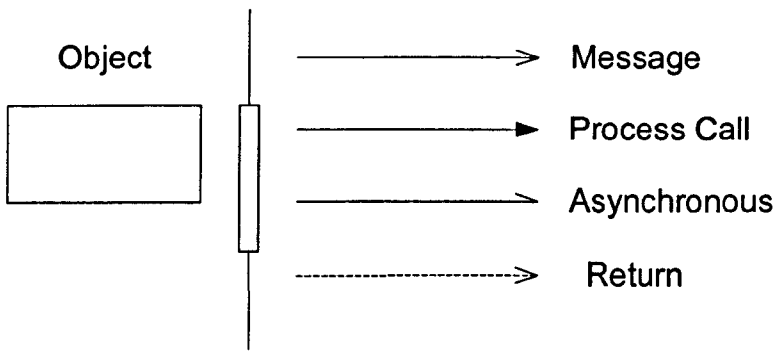

Figure 4.5 Notations for the Process View

4.5 Current Status of CADE Tool

The current prototype implementation of CADE runs on Windows NT 4.0. It is supposed to be an architecture development environment, which primarily supports the component-based control system analysis and design by using the CRC method. It has a visual and interactive user interface and provides a framework in which control system design and associated information can be captured, viewed and modified easily and quickly. It is an open environment; it is easy to adopt new notation and new methods. Currently, CADE generally includes three parts: graphical editor, table editor and relative facilities. The graphical editor provides the service for constructing conceptual design diagrams by adopting the whole UML notation and the extra notation for the control system design. Although a diagram is the best way to present the idea, constructing a diagram is a tedious job. The graphical editor provides a set of

facilities to help the user to reduce the required efforts. The table editor provides some services to generate reports. Through interacting with the graphical editor, it can retrieve the information from diagrams, and generate a framework for the user to edit further. Special facilities are embedded within the environment, which automate all the mundane clerical tasks. The following part describes the CADE features in detail.

4.5.1 Flexibility

CADE has deliberately been made very flexible and informal in the sense of not imposing any unnecessary restriction on when information should be entered, or the order in which things should be done. The aim is not to enforce strict conformance to any pre-defined work-plan but to allow the user to work in the style that he is comfortable with. For example, it is not required to complete the specification of one component before starting another one. The user is free to leave any part of a design partially completed and return to it later. This makes it easier and quicker to record ideas and information. In addition, unlike many other CASE tools which impose a strict top-down decomposition approach, CADE allows the design and analysis to be carried out in a top-down, bottom-up or middle-out fashion, or even a combination of all three. As the primary objective, CADE supports the CRC method, which is the combination of top-down and bottom-up approaches. Furthermore, the use of the notation is also not restricted. The designer can use any notation in any diagram if he feels necessary. The notation can be customised or introduced, if the designer feels it is a better way to present his idea. Through combining the primary object, the designer can easily build a new notation. For example, the component can be represented by some specific icon rather than the box and text. In some circumstances, as it is very easily comprehended by other people, the new notation can be used as the standard notation.

Because analysis and design inevitably needs to go through several iterations and refinements, CADE has been careful to ensure that any decisions made are easy to 'undo'. This not only reduces the effort required in error-recovery, but also helps to encourage the designer to exploit alternative design solutions.

4.5.2 Performance

Speed of performance directly affects the usability of a tool. Because CADE is not designed as crucial to programs to run in batch mode, from the user perspective, an obvious requirement of an interactive program is to have a response time of a fraction of a second. A sluggish tool not only aggravate the user's frustration, but also tends to break down the user's thought process and concentration. Therefore, considerable care has been taken in the development CADE to ensure that it is responsive to the user at all times. This is crucial because a lot of supporting facilities provided may potentially slow down its performance.

4.5.3 Graphical Layout Support

One of the obstacles to the acceptance of the graphical environment has always been the inherent tedium of diagram manipulation. In CADE, the objective is to enable the desired layout of a diagram to be achieved with the minimum effort on the part of the designer. A comprehensive set of automated aids is hence provided for the editing of the control system architecture design diagram.

Every object has a snap mechanism to connect objects together. Objects can be divided into two types; one is a 2-D object such as a rectangle, a circle, and the other is a 1-D object such as a connector, line. Every 2-D object has a predefined snap point, which is represented as a rectangle enclosing a cross. The user can add more snap points, if he feels necessary. The addition of a snap point is based on the object top-left corner position. This means that when the object has been resized or moved, the snap points automatically follow the object. Correspondingly, 1-D objects have connecting snap. When connecting a 1-D object to a 2-D object's snap point, the 1-D object will automatically connect to the nearest snap point that it found. In the meantime, through justifying its shape, it also avoids any crossover with existing objects. For example, there are three objects, which are rectangle A, rectangle B, and circle C in figure 4.6. If object A does not exist, the connector between B and C will follow the dash-line. Because of the existence of object rectangle A, the connector

changes the shape, and follows black line. This really releases the user from tedious work, when the diagram is more complicated.

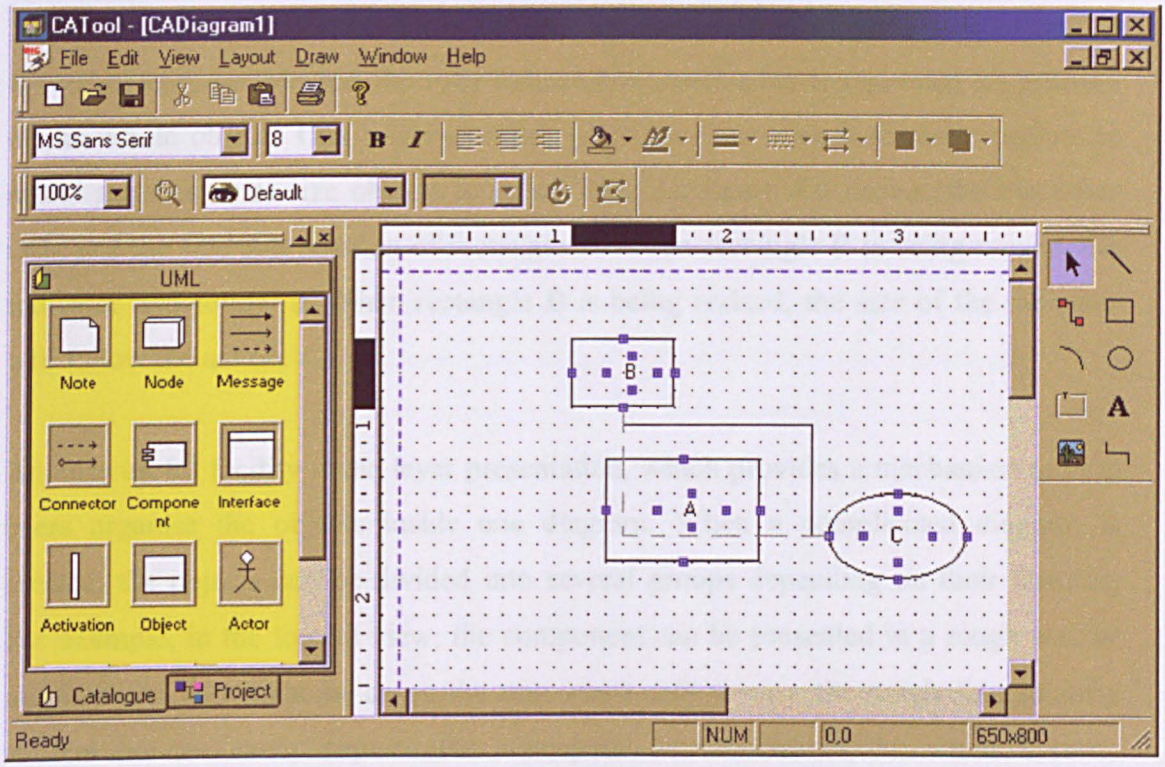

Figure 4.6 The Example of Object Connections

A number of node alignment aids are available to the designer. A flexible facility allows a group of objects to be aligned. The aligning styles include left, right, top, bottom, horizontal centre, and vertical centre. For example, to arrange 4 boxes on a vertical centre, the user first selects the target objects, and then from the layout menu selects "align objects vert. Centre". The horizontal centre position of the first selected object determines the target position of the remaining objects. This action will align all the objects to the same horizontal centre position. The connections in the diagram are automatically cleaned up after this operation.

Other aids for object alignment are the alignment grid and ruler. The alignment grid constrains the placement of objects in a diagram. When an object is created or subsequently moved, its top-left corner automatically snaps to the nearest point in the grid. Therefore, great precision is not required when placing and re-sizing objects. The grid properties can be configured. The user can set the coarseness of the grid.

This depends on the users decision about whether the need is for a small grid size in order to fine tune the position of the objects or a big grid size in order to easily position the object. The ruler is used to indicate the actual position of the object. When no object is selected, the ruler indicates the position of mouse. After one or several objects are selected, the ruler indicator becomes a bar to represent the position of the whole object. This is very useful, when the user wishes to move objects to actual position or re-size objects to actual size. As figure 4.6 shows, the black bar indicates the size and position of rectangle B. When rectangle B is being moved, the indicator will follow it. When rectangle B is being resized, the size of the indicator will follow the action.

Another useful facility is the layer presentation, which provides a mechanism to help users organise the objects inside one diagram. When a complicated diagram is created, the objects can be divided into several groups depending on their features. For example, in the logical view, the component can be presented in a rough way or more precise way. In some cases, the user wants only to view the rough diagram only so that he can easily capture the conceptual architecture of the whole system. Sometimes, the user wants to see the precision diagram only in order to refine the design. The user can use the layer mechanism to divide these two into different layers. By turning on or off the relative layer, the user can very easily obtain the diagram, and does not need to draw several diagrams. In figure 4.7, there are two layers, one is rough, and the other is precise. The left hand screenshot has the precise layer turned off; it gives the designer an overview of the system. When the precise layer turned on, the user can check the details of the interface.

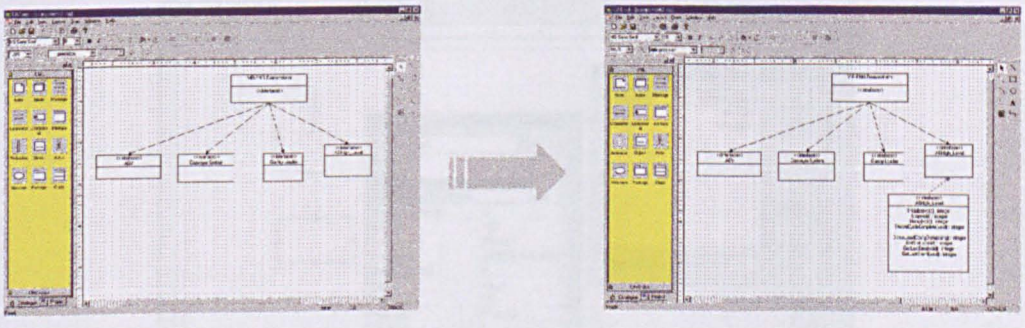

Figure 4.7 The Example of Layer Mechanism

The object property page allows modifying pieces of information, called attributes, which are attached to the objects. Through it, the user can easily modify the object drawing property such as the line width, line style, colour, and fill style. It also allows adding or modifying the information embedded in the object such as comment or functional description. It is vitally important that location and retrieval of an object's attribute be intuitive and straight forward, regardless of the size of the system or the number of defined attributes types. For this purpose, the graphical sheet is the ideal interface. By double clicking the object, the property page will appear. Using selection or input boxes, the user can readily achieve the task. On-line modification gives the user immediate feedback and allows the user to feel the modifying effect. Figure 4.8 shows an example of the property page for connector1. So the diagram that the user can create is not only an attractive picture but also presents information.

For a complicated diagram, normally it is a tedious work to find an object. It would be very helpful if the user could retrieve the created object depending on the name assigned to the object. There are two ways that the user can browse the created objects. One way is to use the object property page. On the top of the object property page, there is a combo box. If the user drops the combo box, he will find all the objects that he created. Another way is to use the object sheet. Through this facility, the user can change the order of objects, rename the objects, and delete the objects depending on the object name. Figure 4.8 illustrates two ways to present the same information. The selection change inside a dialog box will immediately affect the currently selected object.

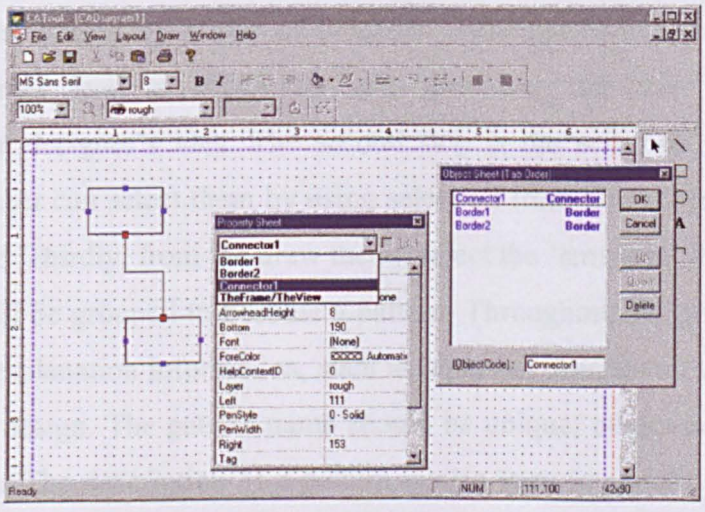

Figure 4.8 The Example of Property Page and Object Sheet

4.5.4 Support for Design Patterns

Just like interface and component reuse, there is another kind of reusability, which potentially may offer even greater benefit to system development. In system development, best practice sharing is a way to ensure which solutions to process and which machine problems in one part of the system are communicated to other parts, where similar problems occur. Best practice sharing eliminates duplication of problem solutions. It extends the idea of software and/or hardware reusability to include the idea of reusable elements of control models by prefabricated modelling solutions to commonly occurring design problems. Central to the description of development best practice sharing is the concept of a pattern. Patterns have been the subject of much discussion in the world of object-oriented analysis and design. CADE provides the facility to create and use a pattern. Actually, the designer can very easily create the pattern himself, and use it to suit the problems he meets.

In CADE, there are several steps to create a pattern. The example of Monitor-Actuator pattern will be used to demonstrate the sequence. Firstly, the designer needs to layout an element structure in an appropriate way, which he believes is the proper diagram to depict the pattern. The use of the notation and the diagram are not restricted; the decision depends on the designer's experience. As figure 4.9 shows, a sequential diagram is adopted to depict the pattern. This pattern includes four elements, controller, monitor channel, control channel, and environment. In order to form a pattern, it is only required to give a proper name for the pattern and names of pattern elements, braced by a pair of brackets. Although names are not restricted, meaningful names lead to efficiently use, moreover, the user can intuitively comprehend the designer's idea. The second step is the selection of the involved elements. The user can select them by using select all from the edit menu, or directly using the mouse. Thirdly, from the draw menu, select the 'template' item, and then all the elements will be grouped to become a pattern. Throughout the property page, the user can input explanation information, such as a pattern description to use as a guide in analysis and design. The pattern name should be unique; otherwise, it will lead to system collapse. The description of a pattern is also very important for encouraging others to use the pattern. Finally, the pattern is stored by means of drag-and-drop the

designed pattern to the catalogue. The user can give an abstractive icon to the catalogue. Now a new pattern has been created and is ready to be used.

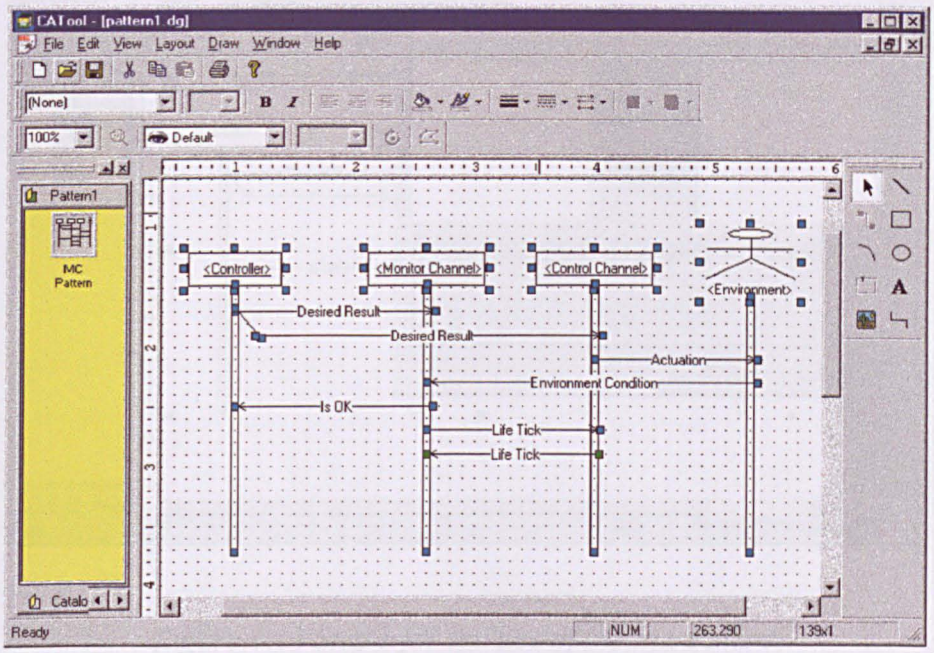

Figure 4.9 The Example of Creating Monitor-Actuator Pattern

Compared to create a pattern, reusing a pattern is quiet simple. There are three steps during the process, guided by a wizard, which are illustrated in figure 4.10 and 4.11. It starts with dragging and dropping the selected pattern into the diagram. After that, the first dialog box consists of the essential elements, the optional elements and the description of the pattern. The description of the pattern, at the bottom of the dialog box, provides some basic information (such as the design purpose, involved relationships, e.g.) for the user to initially check whether it is the right one or not. While the essential elements are the basis for the use of the pattern, the optional elements need to be selected by the user. In this example, the optional element is empty; it does exist in some other patterns. The second dialog box asks the user to assign the proper name for the specific element. In this example, the environment in this implementation is a mechanical part, with which the controller interacts.

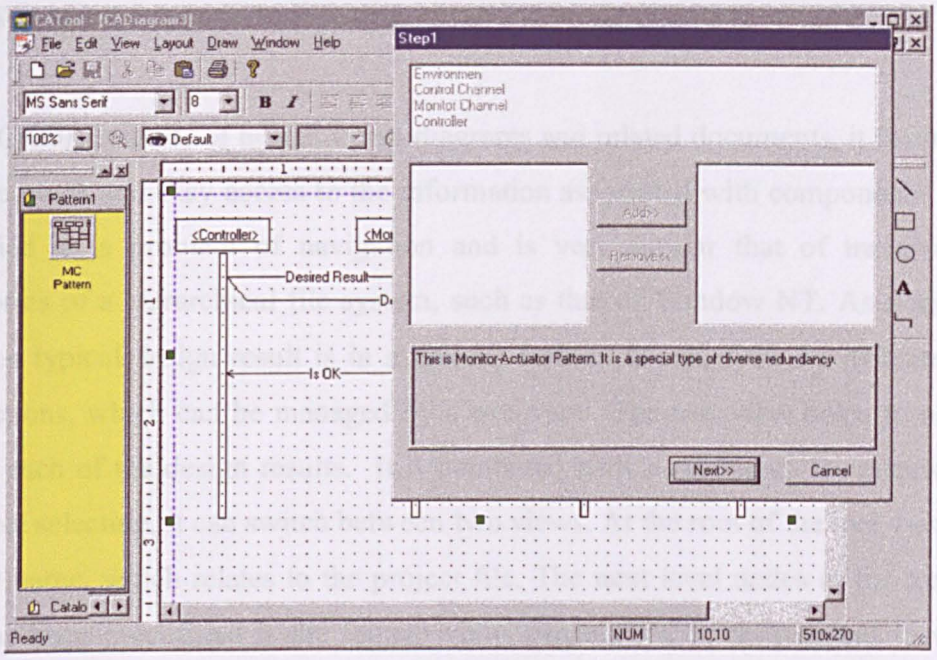

Figure 4.10 Step 1 for the Use of the MA Pattern

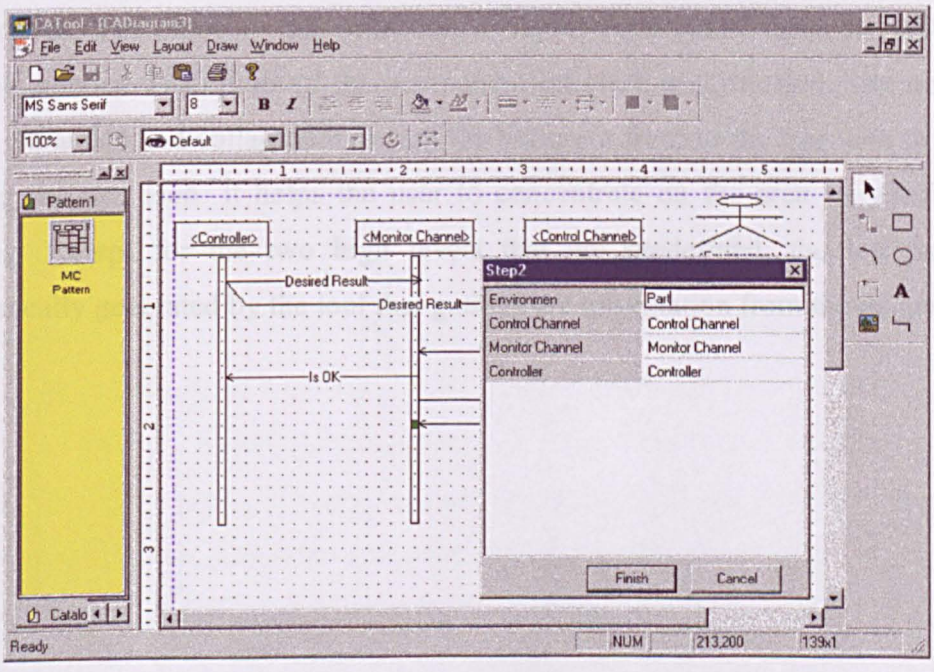

Figure 4.11 Step 2 for the Use of MA Pattern

4.5.5 Support for System Navigation

Since CADE consists of nested views diagrams and related documents, it is important to have quick and easy access to the information associated with components. This is classified as a problem of navigation and is very similar that of traversing the directories of a hierarchical file system, such as that of Window NT. As a matter of fact, the typical design result is in a layered fashion to represent the different level abstractions, which can be managed by a tree view. The tree view helps to navigate within each of the design results. It is combined with a catalogue. By means of the tab sheet selecting, it can switch between two views. At the root of the tree view is the project name, which relates to the project file. The next level nodes of the tree view that has been predefined is the logical view, process View, deployment view, task view and report, which corresponds to the CRC method. Task view is for collecting the diagram that relates to the task analysis and description. The other three views are to gather the diagrams of the components interface specification from three different perspectives. The report is the collection of report documents; there are tables that give the detailed description of the component interface specification. The tree view can be expanded and collapsed by using the button in front of the tree icon or double clicking the tree icon. It helps the user to concentrate on the area on which he is working. Except for the two high levels that are predefined, the left parts are automatically generated by the tool that gathers the information from the diagrams.

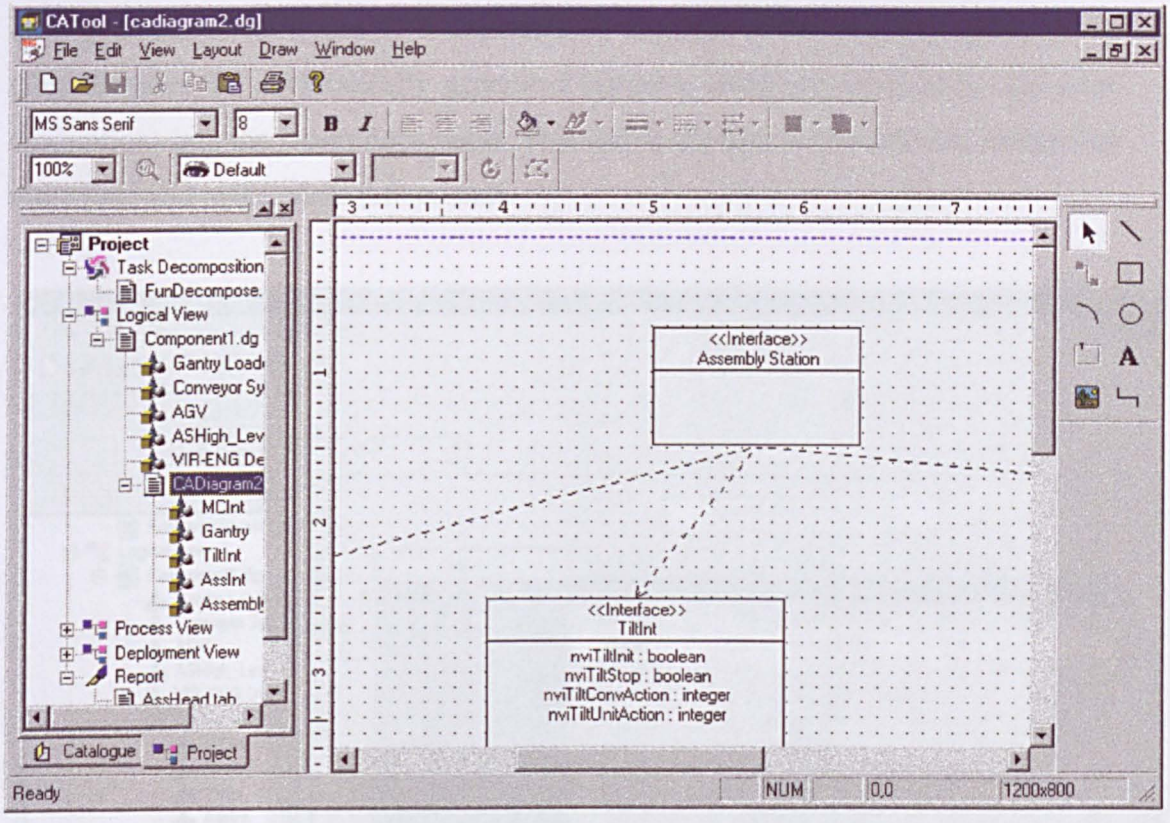

Figure 4.12 The Example of System Structure Tree

4.5.6 Report Facility

As an analysis and design environment, the documentation of the result is essential for CADE. The primary objective is to gather and summarise the information generated during the analysis and design process in a coherent and appropriate way. Consequently, the content of a report is automatically extracted from the information embedded within the generated diagrams and from the output Word and/or Excel format documents. Basically, the reports include the component list, the component interface specifications, the relationships among the components, and the constraints in table form. By default, the report facility defines the template needed and where to store the resulting document(s). It can automatically layout content, synchronise existing content and create new ones. In addition, the default layout of the report can be customised; the user can add in more columns and/or rows, in order to present sufficient information. Since the Word format is editable by many popular commercial word processors, the contents and layout are editable after the report has

been generated. The figure 4.13 illustrates an example of the interface specification table. Besides the automatically generated contents, users are required to add some descriptions into the report documents. This forces the user to consider the design one more time and make some refinements.

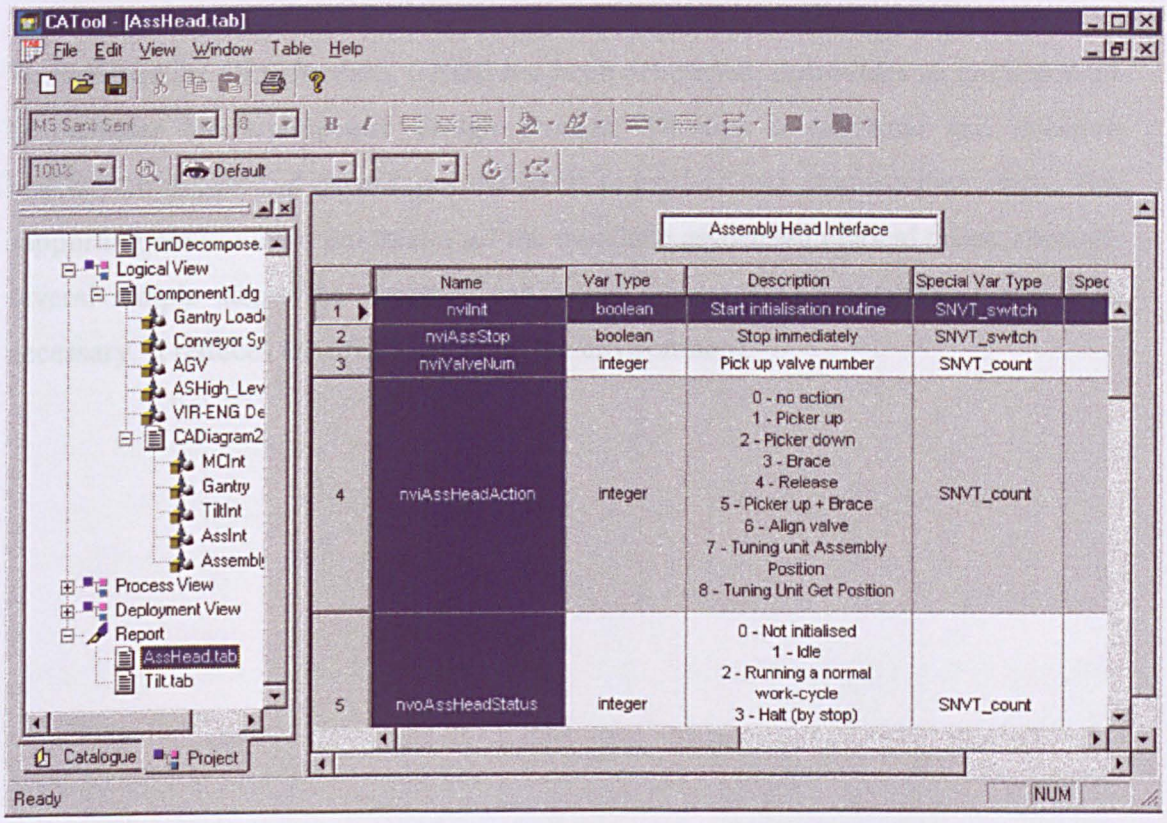

Figure 4.13 Table Sample

4.6 Summary

Many current reported works are focused on component-based architecture analysis and design with an emphasis on addressing the software aspects. However, existing approaches do not consider the features of the machine control system development lifecycle. In order to cope with the features of machine control system, CRC has been proposed, which converts the control system requirement into the architecture design by means of using four activities. Inspired by the software analysis and design method, it adopts and extends the UML notation to satisfy control system characteristics. However, the difference from pure software development approaches

is that CRC emphases hardware component and software component co-design. In addition, the approach emphases the communication between control engineers and mechanical engineers by means of sharing common knowledge. It ensures that both groups consider the complete machine-system development strategy (e.g. such as machine re-configuration and maintenance from each perspective, etc.).

To facilitate the CRC method, CADE has been developed. It provides an environment to facilitate the architecture design process. Through a graphical and intuitive interface, it assists and encourages designers to explore and examine their ideas. The supporting environment automates all the mundane associated clerical tasks. Through several simple steps, the designer can adopt a new set of notations when it is necessary. It reduces the effort to extend the environment.

Chapter 5 Control Logic Component Design and Implementation

5.1 Introduction

Following the proposed control system development track, the objective of this chapter is to facilitate the detailed design and implementation of control components generated in the control architecture design. The accomplishments of the control architecture design described in the previous chapter can be summarized as designing abstract components be means of defining component interfaces and interconnections between components including components' interactions and architecture. In the context of component-based development, the task of this chapter is to transform abstract components into concrete components. It includes the detail design for abstract components and the implementation of abstract components.

Primarily, UML is adopted to facilitate the control architecture design and represent the component interface specification. However, UML is a generic, semi-formal, incomplete language and lacks an execution model. In order to be useful in the detail design, UML needs to be transformed into a formal language. The IEC 1131-3 standard is proposed to carry out the detail design, which includes formal and complete languages to create designs that can be checked for accuracy. This can provide accuracy verification and validation in early development phases, in contrast to UML conceptually verifying design. Such capability comes from using sequential function charts (SFCs), which are state-transition diagrams derived from Grafcet (David and Alla, 1992). They are very expressive for describing, in a readable and intuitive way, sequential behavior of complex systems. However, before the detail design can be processed, the transformation between UML and IEC 1131-3 needs to be developed to cover the transition from conceptual design to detail design.

In the context of component implementation, software component-based development (CBD) has become a popular topic in software engineering (Brown and Wallnau, 1998). Meanwhile, it has been introduced into industry. In particular, the basic

philosophy has been widely investigated to take advantage of its basic concepts (abstraction, encapsulation), which showed effectiveness in increasing modularity and reusability (Duran and Batocchio, 1994; Maffezzoni *et al.*, 1999). One attractive feature of CBD is the emphasis on standardizing interfaces between components, without restriction on how the implementation is accomplished (Szyperski, 1998). Subsequently, CBD is closely related to design in separating interface and implementation, which determines that components do not relate to specific languages. Theoretically, it can be produced in any language and used by any language.

However, while components are consumed and produced in the software engineering field, current practice for developing machine control systems is inefficient and often few components (e.g. electronic and software parts) can be reused "as it is" in subsequent projects, increasing overall machine system costs. One major obstruction limiting the adoption of component-based development of software in the industrial control area is the lack of viable component development languages for control engineers. Although there are a lot of object-oriented languages such as Java, C++ and component-oriented languages such as Component Pascal, the 'learning curve' of these languages is often 'too steep' for control engineers. Typically, a control engineer has a good knowledge of the controlled process and his main role is limited to designing the related control sequence. In addition, there is not yet a standard tool, which is sufficiently powerful, versatile and simple to use, and with which it is possible to carry out formal analysis of correctness.

Based on the argument that the component implementation does not rely on specific languages, this chapter proposes a systematic and easy way to analyse and construct control components using the IEC 1131-3 standard. It was not only widely recognized languages in industrial control area, but is also widely accepted by control engineers. Moreover, the IEC 1131-3 does have some advantages beyond most general programming languages. As stated earlier, the IEC 1131-3 is capable of supporting the detail design, which practically closes the gap between design and implementation.

However, five languages of IEC 1131-3 are not object-oriented or component-oriented languages, which are the typical programming languages for CBD. Historically, as originally developed, the languages were not designed for CBD. This chapter describes the methods and the tools used to extend the basic languages and develop underlying mechanisms within IEC 1131-3 language to facilitate component-based development. The methods introduced have little or no impact on the basic languages. It is essential for their users that the languages maintain the feature of simplicity to learn and use.

The chapter is organised as follows. Section 2 introduces the background, which includes the requirements of component-based development and some basic definitions. Section 3 addresses the structure of the Control Logic Programming Environment (CLPE), and the control component development process via CLPE. The detail design method via the IEC 1131-3 standard is introduced and formalised in section 4 and the implementation of control components is described in section 5. Sections 6 and 7 introduce the underlying mechanisms and associated tools to facilitate control component development. Section 8 describes the results of evaluating the implementation performance. Finally, in section 9, the results are summarised.

5.2 Background of Control Component Development

Before discussing the proposed approach to facilitate component-based development, this section explains some concepts related to this research.

5.2.1 Requirements of the Integration Infrastructure

The component-based development process relies on a variety of integration services provided by the infrastructure to interoperate components. The use of an integration infrastructure enables designers to select and configure the use of appropriate groupings of components to achieve the designated purpose of the system. The

essential services provided by the integration infrastructure can be defined as the composition operations, which are summarised as:

- **Supporting multiple communication channels.** Every channel allows components to independently exchange data and invoke methods, which could have different priorities. A good example is the message mechanism in the Windows operation system.
- **Higher-level component aggregation.** This operation combines components to produce higher order functional constructs. The combination typically involves creating a hierarchy of components. Although the aggregated components behave as one, the components execute independently; they may not be started and terminated at the same time.
- **Recursive component composition.** Component composition creates a new component rather than an application. This is a powerful technique that enables components to become software abstractions at different levels, and provides the bottom-up progressive development strategies. It plays an important role in providing scalability to the component. The composed components must be executed dependently; they are started and terminated simultaneously.

5.2.2 Explanation of the Terms

Some terms are specified in this chapter, which can be summarized as:

Component

In this chapter, a component is the actual implementation of a component, which is compliant with the VECOM model. A component offers a set of data and services to the consumer. This information is exposed to the consumer as local and/or remote interfaces. Examples of component information and service are simple data (sensor data) and specific operation (actuator operation).

Connection

A connection provides an information exchange channel between the IEC 1131-3 program and components. This includes the means of encoding and decoding

information, and transferring the information between two parts of the elements. Connections do not describe the syntax and semantics of the information being transferred. The information is determined by the component interfaces and the data and functions of relative IEC 1131-3 program. It enables the operation of components as ordinary entities of the IEC 1131-3 program.

Connector

A connector can be considered as a specific component that provides the communication mechanism and media. Connectors facilitate connections; every connector includes at least one connection. The connectors provide the mechanism, which establishes the rules that govern connections and provides relative aided functions.

Port

A pair of ports represents the two ends of a connection. At the one end, the IEC 1131-3 port plugs into an IEC 1131-3 program, feeding in and pumping out information. At the other end, a component port plugs into a component, making contacts with the component. The ports are classified as data ports and method ports. A data port connects to event, message, and continuous data stream within the component and connects to variables in the IEC 1131-3 program. While the method port adopts the component method, it connects to Function Block (FB) in the IEC 1131-3 program.

Adapter

An adapter is considered as a wrapping mechanism, which converts the proprietary interface and/or communication protocol into a standard component interface. Primarily, it processes the internal information. In addition, it also includes the basic functions to manage the established connections.

5.3 Overview of Control Logic Programming Environment (CLPE)

5.3.1 CLPE Structure

Conceptually, the functions of the CLPE underlying mechanism can be identified as the use of existing components and the production of new components. As shown in Figure 5.1, the key part of CLPE is an implementation of the IEC 1131-3 standard, which is ISaGRAF in this research. As underlying mechanisms, the connector at the right hand side provides services to enable the IEC 1131-3 program to consume components; the adapter on the left hand side provides services to wrap IEC 1131-3 programs into components. The communication between the IEC 1131-3 program and the connector is through shared memory; the communication between the IEC 1131-3 program and the adapter uses the ISaGRAF specific communication interface. It is worth stressing that the data communication of the underlying mechanisms adopts data push architecture by means of combining publish/subscribe techniques, which is recognised as an efficient architecture for time constrained communication (Kanitkar and Delis 2001). It has already been defined in the VECOM interface and realised by the implementation of connector and adapter. In order to promote and simplify the development process, two tools have been developed namely VCon and CBuilder. VCon aims to configure the connector based on the component interface, so that the connector establishes the communication with components. CBuilder aims to instantiate the adapter according to the implementation of the IEC 1131-3 program.

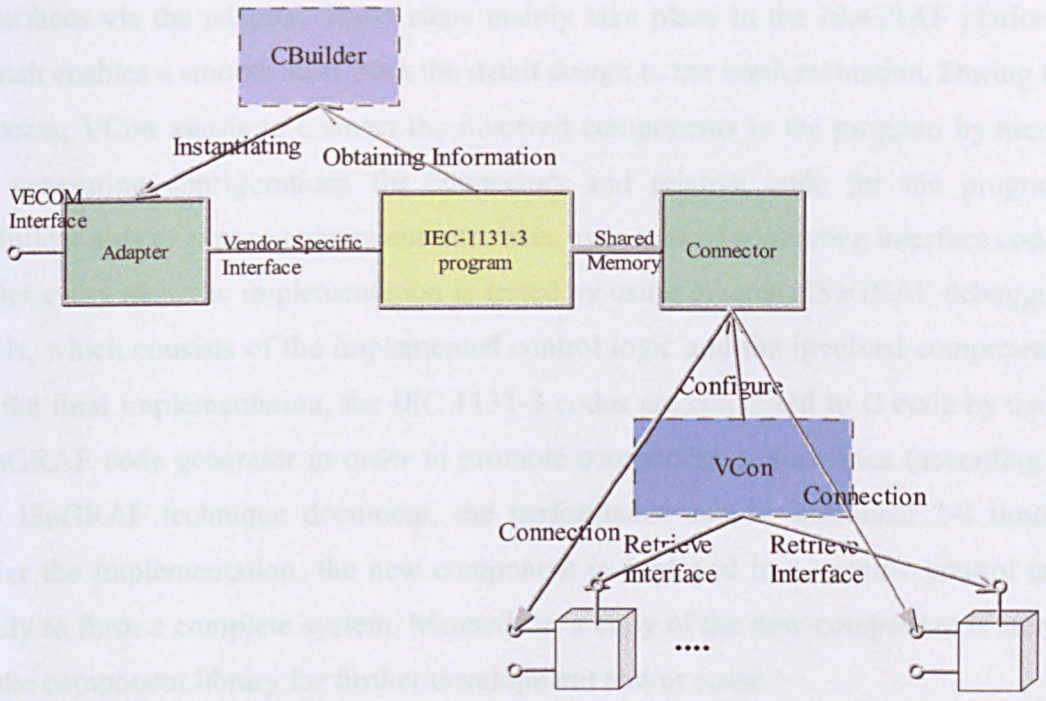

Figure 5.1 Overview of CLPE Structure

5.3.2 The Development Process within CLPE

Figure 5.2 depicts the entire CLPE tools position in the component development process. Conceptually, it can be split into two parts, which are the design phase and the implementation phase.

During the design phase, the designed abstract component from the control architecture design imports into CLPE to carry out the detail design. The design actions take place in ISaGRAF mainly via SFC, which specifies how to implement abstract components. The detail design commits to specific technologies, which determines the internal structure of components, involved components, exception handler and sequence to facilitate designed behaviours. Furthermore, the detail design involves more hardware dependent design than the architecture design.

In the implementation phase, there are three steps to construct designed components. It begins with coding the control components, continues with composing or aggregating components via connectors and finishes with exposing component

interfaces via the adapter. These steps mainly take place in the ISaGRAF platform, which enables a smooth shift from the detail design to the implementation. During the process, VCon assists to connect the involved components to the program by means of generating configurations for connectors and relative code for the program; CBuilder aids to expose component interfaces by means of generating interface codes. After every step, the implementation is tested by using inherent ISaGRAF debugging tools, which consists of the implemented control logic and the involved components. In the final implementation, the IEC 1131-3 codes are converted to C code by using ISaGRAF code generator in order to promote component performance (according to the ISaGRAF technique document, the performance can be improved 2-8 times). After the implementation, the new component is deployed in a specific control unit ready to form a complete system. Meanwhile, a copy of the new component is stored in the component library for further development and/or reuse.

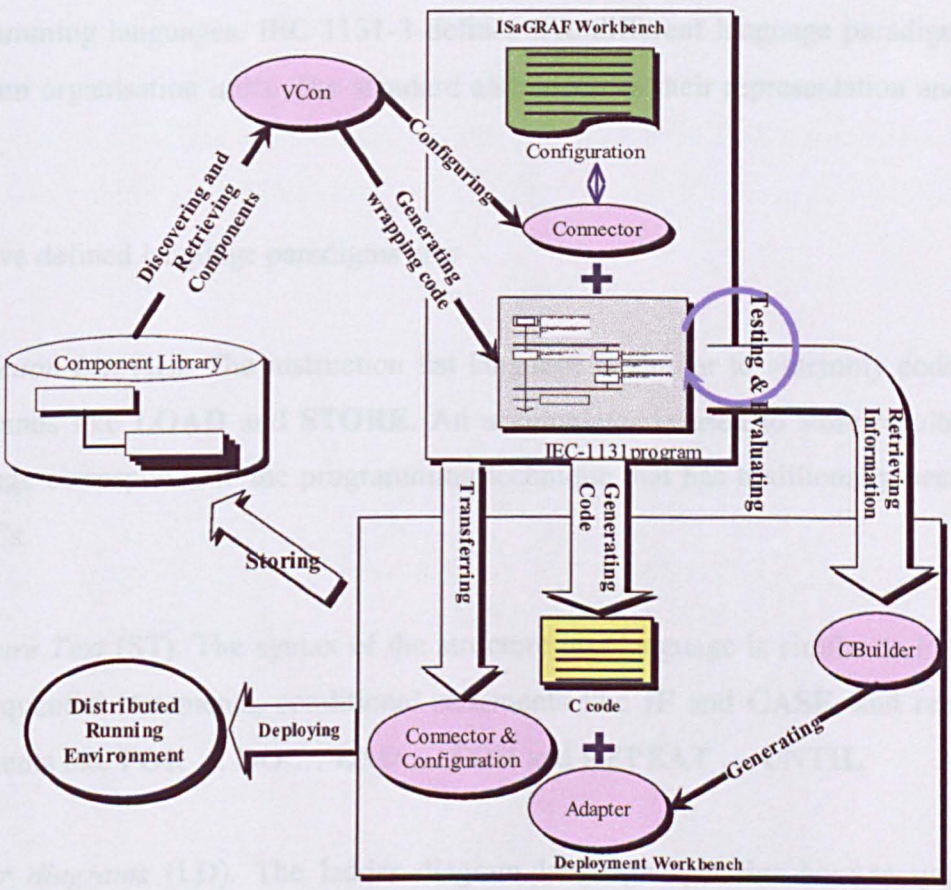

Figure 5.2 The Development Process

5.4 The Detail Design via IEC 1131-3

Since IEC-1131-3 languages have been accepted and used in the industrial control area for several years, there are intensive research efforts around using the IEC 1131-3 language to implement control systems (Zaytoon and Carre-Menetrier 2001). Before the discussion of the detail design, it is necessary to explain the IEC 1131-3 languages. The detail design includes the design process and the general design principles associated with IEC 1131-3 languages.

5.4.1 IEC 1131-3

The IEC 1131 is standardised by the International Electrotechnical Commission (IEC), which contains five parts. Part 3 is mainly concerned with describing the programming languages. IEC 1131-3 defines five different language paradigms and program organisation units. The standard also specifies their representation and rules of use.

The five defined language paradigms are:

Instruction List (IL). The instruction list language is similar to assembly code, with commands like **LOAD** and **STORE**. An accumulator is used to store results. This language corresponds to the programming technique that has traditionally been used in PLCs.

Structure Text (ST). The syntax of the structure text language is similar to Pascal. It has sequential statements, conditional statements like **IF** and **CASE**, and repetitive statements like **FOR** ... **DO** ... **END** ... **FOR** and **REPEAT** ... **UNTIL**.

Ladder diagrams (LD). The ladder diagram language specifies how to use relay ladder logic diagrams to implement Boolean functions. This is a common language in modern PLCs.

Function block diagram (FBD). In the function block diagram language, all functions, input and outputs are represented as graphical blocks. They are connected by lines representing the data flow. The direction is always from left to right, except in feedback paths.

Sequential function chart (SFC). SFC consists of two main elements, steps and transitions. Steps can be active or inactive. The state of the SFC is determined by which steps are active. A transition has a Boolean input, the transition condition, which can be described by any of the above four languages or by a Boolean variable. A transition will be fired when the step above it is active and the transition condition is true. This language describes parallel and synchronised sequences of elementary operations, shown in figure 5.3. It allows for explicitly describing the concurrent process with due consideration to the system response.

Based on the above description, IL and LD are primitive languages, which are not appropriate for component-based development. They are intended to represent low-level physical elements, such as relay and physical memory, which are neither proper nor allowed in the operation of components. Therefore, they are not recommended for use.

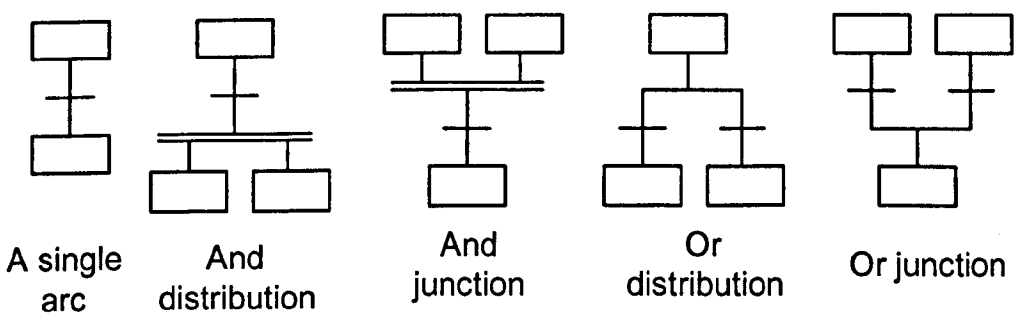

Figure 5.3 Basic Operations of SFC

5.4.2 The Detail Design of Control Components

The detail design takes place in the IEC 1131-3 space. The link between the architecture design and the detail design is the transformation. The process is not only

a simple mapping between two schemas but also includes a decomposition of the design tasks. Because of the lack of team development support in IEC 1131-3, the architecture design results are divided into several IEC 1131-3 projects, so that the design team can carry out work in a concurrent fashion, which is more important in complex systems. Such practice is based on the structure of components. It should also be concerned with the coupling level of components; closely coupled components should be included in one project.

5.4.2.1 Component Representation in IEC 1131-3

One key issue to map between the architecture design results and the IEC 1131-3 standard is how to represent components within IEC 1131-3. Actually, there are several ways this can be achieved.

The IEC software architecture uses the concept of program organisation unit (POU) as building blocks based principally on modularity and encapsulation. A POU is one of these three types: function, function block (FB) or program. The behaviour and structure of POUs are defined by a type declaration. POU does encourage a certain degree of software reuse, from the macro level with programs to micro level with function and function blocks. In particular, FB supports certain level information hiding. It allows defining internal variables even with the same name. The interface of a function block, the input and output data flow of the function block, can also be separately defined without worrying about conflicting. Therefore, it can be used as one representation of a component. However, it is only suitable for representing simple components; it is inefficient to handle for a complex component.

In IEC 1131-3, there is another way to represent a component, which is the project unit originally representing the unit for a single PLC. In ISaGRAF, it is named as the resource unit to represent this concept. Inside the resource unit, the programs are organised as in traditional IEC 1131-3 and support every IEC 1131-3 language. Among the resource units, they are isolated; that means they can freely define the variable and program elements without conflicting. Therefore, the resource unit can effectively handle complex components. However, such an approach requires more

computing resources than the basic units of IEC 1131-3, because it is regarded as a complete project unit. For a simple component such as a PID controller, the better choice could be a FB.

5.4.2.2 Static Structure

The transformation begins by converting the static structure in the control architecture design to a representation in by the deployment and logic views. In the architecture design, the components involved in the whole system are considered. Actually, some components could use off-the-shelf components as far as they confirm the required interfaces, or are not implemented in software aspects. For example, a standard PID algorithm could have already been built or purchased in the market; a sensor component is not related to this implementation. Therefore, the identification of the components is the initial action before the actual design is carried out.

After the identification, components are translated into individual resource units or FBs. The name of the resource unit or FB normally follows the component name. The related interface of the component is defined in the resource unit using variables or functions. In fact, this step just defines a skeleton of the program; the real inside code will be developed afterwards. The static relationships between components can be translated using the resource unit binding. The example is shown in figure 5.4.

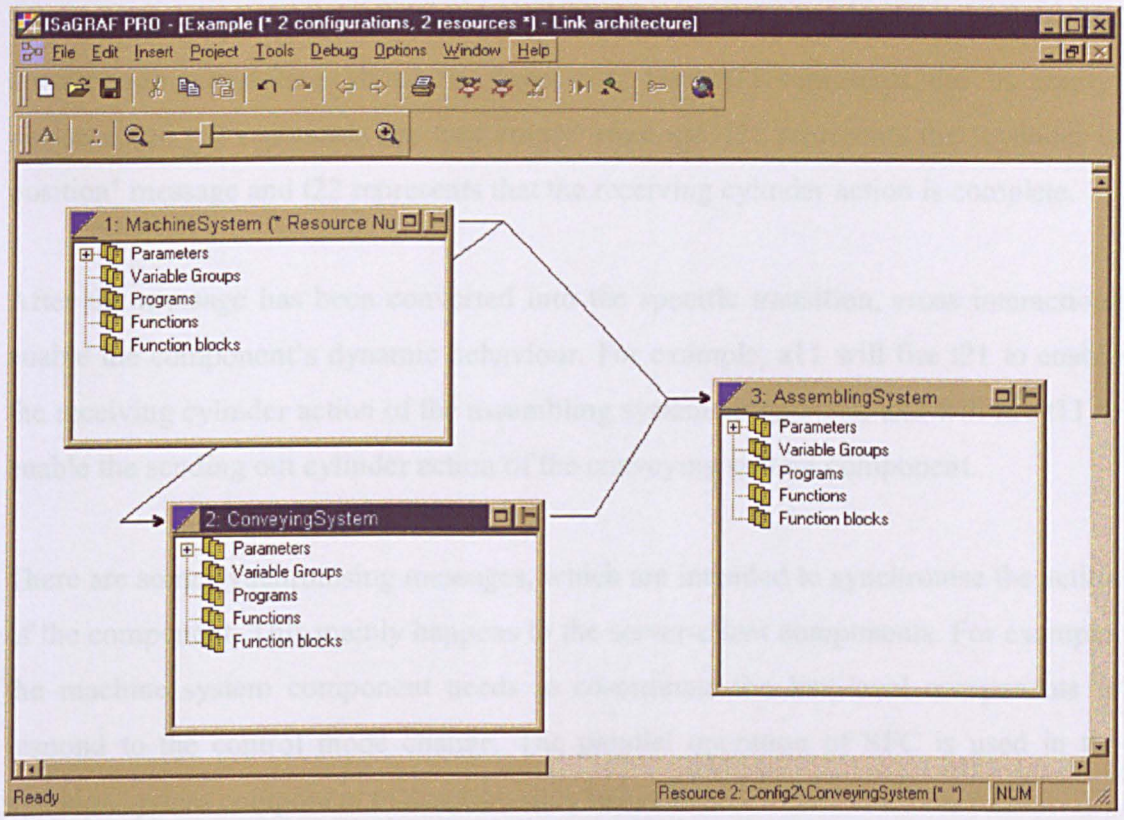

Figure 5.4 An Example of Component Representation

5.4.2.3 Dynamic Structure

Besides the static structure, component specifications include the dynamic behaviour represented in the process view and mainly associated with the sequential diagram. Basically, a sequential diagram describes the group behaviour of component, which can be efficiently replaced by using SFC. During the translation process of the static structure, every component is transformed into an independent resource unit. Consequently, every component has at least one SFC. In some cases, there may be more than one SFC in one component, so that the implementation can be modular and easy to debug. The dynamic behaviour of the component should be in the main SFC.

The sequential diagram consists of messages and actions. In order to construct the SFC, every message passing point or process call point needs to be identified and translated to the transitions of the SFC. Figure 5.5 illustrates the relationship between the conveying system component and assembling system component. The messages

are 'cylinder in position', 'is empty', and 'not empty'. They can be translated to the corresponding transition shown in figure 5.6. Here t11 represents the 'is empty' message and t12 represents the 'not empty' message. t21 represents the 'cylinder in position' message and t22 represents that the receiving cylinder action is complete.

After the message has been converted into the specific transition, cross interactions enable the component's dynamic behaviour. For example, a11 will fire t21 to enable the receiving cylinder action of the assembling system component; a22 will fire t11 to enable the sending out cylinder action of the conveying system component.

There are some synchronising messages, which are intended to synchronise the action of the component. This mainly happens to the server-client components. For example, the machine system component needs to co-ordinate the low-level components to respond to the control mode change. The parallel operation of SFC is used in the machine system component to describe such behaviour.

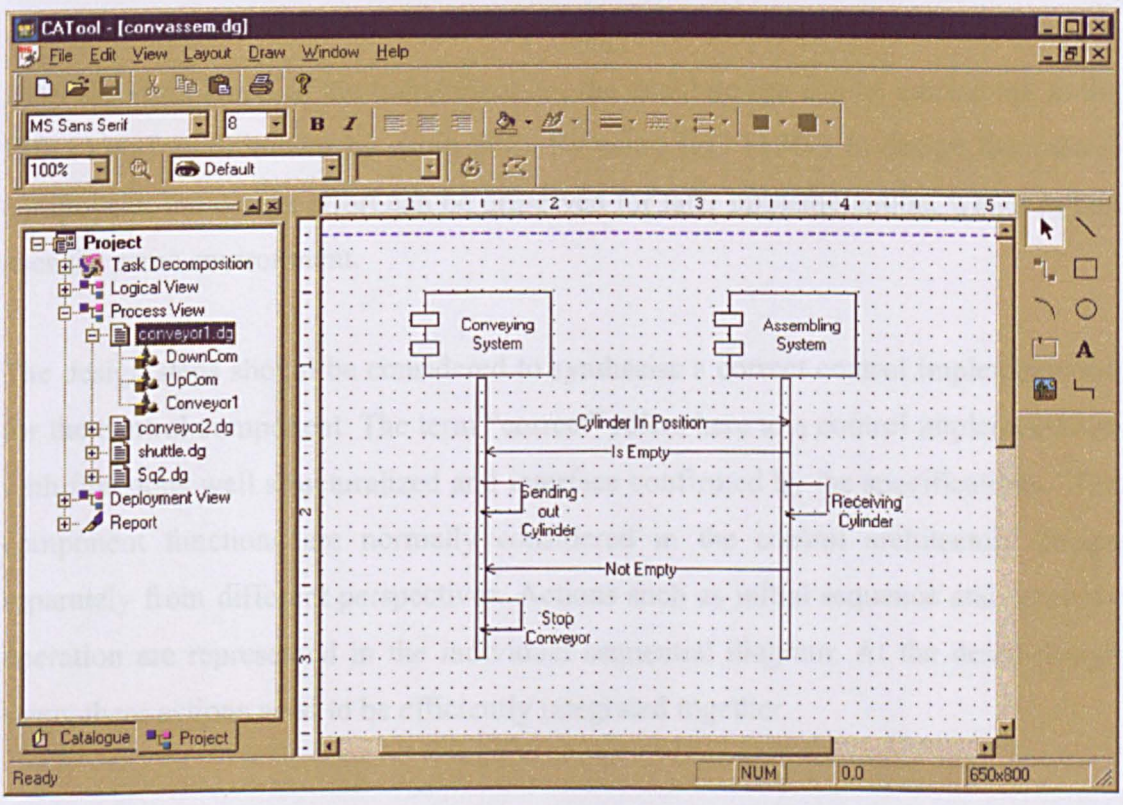

Figure 5.5 The Sequential Diagram Example

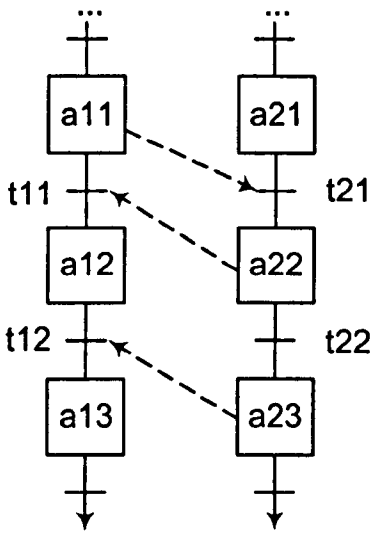

Figure 5.6 The Corresponding SFC

5.4.2.4 Functional Design

After the completion of the transformation, the detail design can be carried out in the IEC 1131-3 environment by using SFC. By using IEC 1131-3 to design the control component, the design effort can be preserved for later implementation, which in fact uses the same environment.

The design steps should be considered to synthesise a correct control implementation for the control component. The term "correct" refers here to a control implementation with functions well structuralized and interface confirmed by the specifications. The component functions are normally considered in the control architecture design separately from different perspectives. Actions such as initial sequence and recovery operation are represented in the individual sequential diagram. At the detail design stage, these actions need to be efficiently integrated together.

A framework for the component is proposed to systematically locate functions, which consists of underlying function, initial action, recovery action, mode-changing action, and error detection action combined with normal operation action as shown in figure

5.7. The underlying function is a pre-defined part specified for the "Connector" operations, which consists of S1, T2, and S3 in the figure and is kept the same over all design and implementation. Nevertheless, it is not part of the design activities. Eventually, the assistant tool VCon automatically generates the handling code based on the connection configuration, which will be introduced later. The initialising action is the block, which includes actions to set up the necessary initial status such as turning on the power, turning on the air. In addition, by default, the initial operation mode should be manual. A waiting stage is for waiting for the mode to change from manual to auto. When the control mode is changed to auto, the transition is fired, and SFC will step to the normal operation. As the figure shows, recovering action uses an *'or distribution'* connection to connect to the waiting transition alongside the normal operation actions. When errors occur, it should include actions to restore the system. The error detection function is S41 running in parallel with the normal operation action; it will be activated when the system turns to automatic mode. After the system turns to manual mode, the detection function will be terminated. The other normal operations are stopped as well. Subsequently, the system goes through initialising action and back to the waiting transition as the start point of the manual mode. The final part is the shutdown operation.

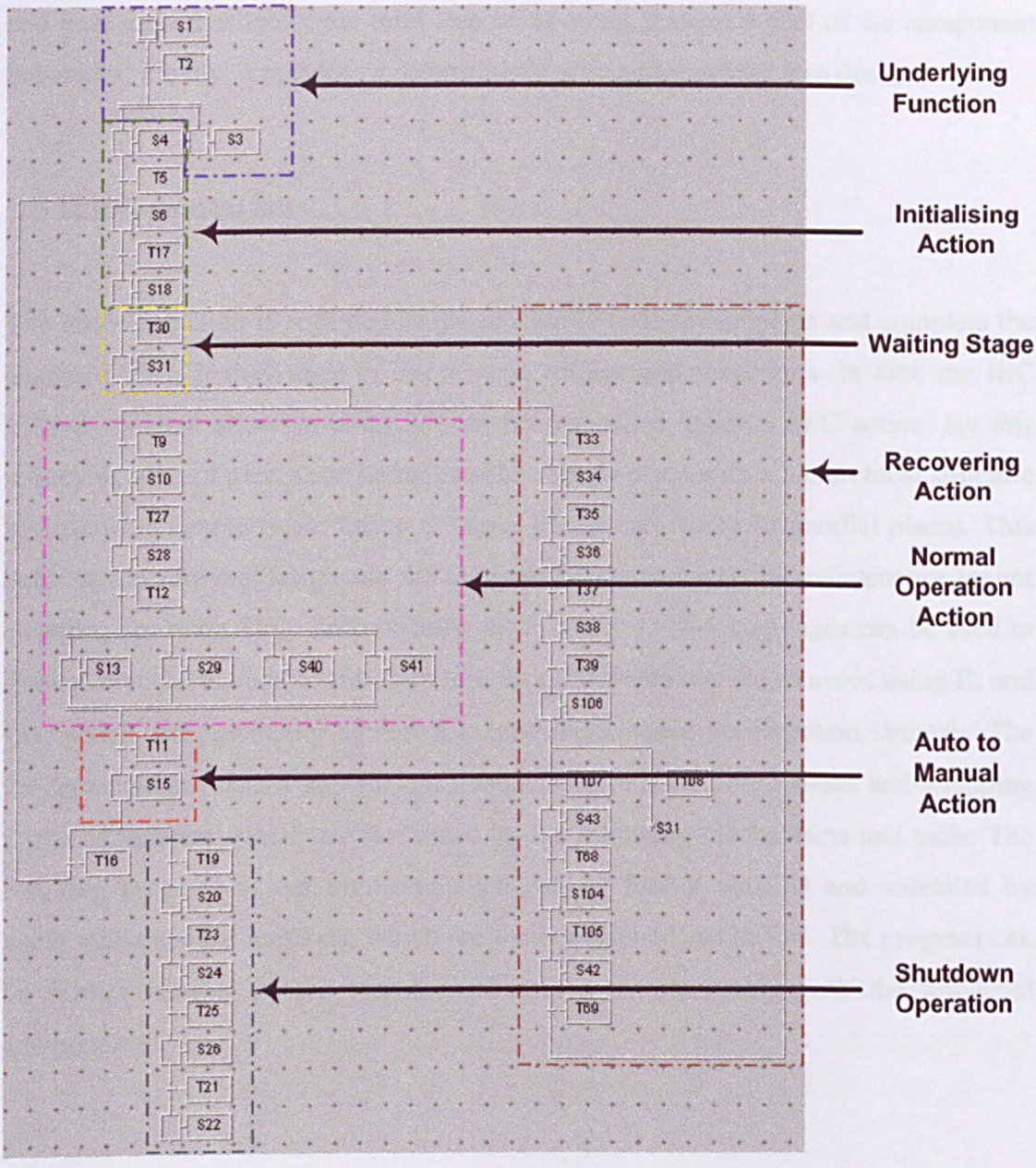

Figure 5.7 The Framework for the Detail Design

Based on the proposed framework, the component control functions can be properly organised. Then the main detail design shifts to design the normal operations in the automatic mode. It can refer to the standard discrete event-driven system design process, which extends the normal operation action block. In the automatic mode, the first step is to identify stable situations during the operation process. Then, the largest permissible behaviour, with respect to a number of given safety constraints, should be extracted. The results may exhibit a number of deadlocks in execution. To identify

and treat these deadlocks, the third step relies on an abstract model of the component behaviour. Finally, it provides a deterministic, safe and deadlock free design.

5.5 Implementation

The implementation is regarded as the process to code components and complete the designed SFC. It may need to insert some actions and transitions. In fact, the IEC 1131-3 standard allows a child SFC to be embedded inside a SFC action. By this reduction, a set of places can be reduced to a single place with a hierarchical structure by following simple rules: fusion of series places and fusion of parallel places. This helps to separate the design and the implementation and ease the maintenance by not changing the main SFC. Theoretically, all five IEC 1131-3 languages can be used to implement control components. But it is strongly recommended to avoid using IL and LD, which are potentially against the component-based development strategy. The implementation process also includes connecting to other components and wrapping into a component, which are facilitated by the following mechanisms and tools. The resulting program of the implementation can be further verified and validated by using mathematical methods, which are well developed in Grafcet. The program can be further verified in the virtual environment by combining with the low-level components.

5.6 Connector

Allowing IEC 1131-3 programs to consume components is a key issue addressed by CLPE. One intuitive solution is to extend the basic set of IEC 1131-3 languages. However, the extension of IEC 1131-3, such as structure text (ST) resembling PASCAL, violates the adoption basis of the standard. Actually, such a practice is not much different to asking a control engineer to learn a new object-oriented language. It is evident that object-oriented programming languages have a much steeper learning curve and are more difficult to comprehend. Nevertheless, the modification of IEC 1131-3 languages could cause portability problems and lose simplicity of languages. An alternative approach is to regard the IEC 1131-3 languages as media programming languages. After coding, the codes convert to a general language, such as C, C++. There are many mature tools to adopt the generated codes into the standard component model. However, one big advantage of the IEC 1131-3 languages has disappeared, which is visual debugging and testing program. Furthermore, since the process is not reversible, the continuity is broken. Thus, this is not a good solution to the problem.

In order to solve the above problems, a connector mechanism is proposed to establish a bridge between IEC 1131-3 programs and components using the standard extending mechanism. Such practice allows IEC 1131-3 programs to use software components, but not to lose the advantages of IEC 1131-3.

5.6.1 Objectives

The objectives of developing the connector are:

- Generalisation. A connector should not be developed for only one specific IEC 1131-3 workbench. It should only rely on the IEC 1131-3 standard.
- Organisation. An IEC 1131-3 program cannot directly invoke the used components. So the connector has the responsibility to both initialise and destroy the components' instance. Once connection is established, the connector also

needs to monitor invalidation of the connections. If a connection is not available, the connector should produce sufficient error messages for the IEC 1131-3 program, because of its unawareness of components.

- Stable. Connectors should be loosely coupled with IEC 1131-3 programs or components and can be upgraded without modifying developed IEC 1131-3 programs. When some bugs in a connector have been fixed or new auxiliary functions have been added in, replacement should only relate to the connector and no further modifications should be necessary.
- Configurable. The operations are facilitated by means of configuring a connector without requiring the program. Furthermore, changes to composed components or insertion of new components can also be facilitated by re-configuring the connector.
- Real-time. Connectors should confirm the timing constraint on a real-time task. This is the deadline, that is the time before which the task should complete its execution. In the context of control systems, all task deadlines must be met. Here, it is assumed that IEC 1131-3 programs and components already fulfil the real-time requirements.

As outlined above, the first requirement is specifically for IEC 1131-3 programs. The next three are general requirements; they should be addressed in any similar developments. The last one is control system specific. The following section will describe the implemented connector satisfies these requirements.

5.6.2 Connector Structure

The participants in the connector include connection manager, configuration storage, common application interface, facility, IEC 1131-3 port and component port. The static structure is illustrated in Fig 5.8.

The connection manager takes charge to create connections between the IEC 1131-3 port and the component port. Once a connection is created, connection manager keeps track of the connection until it is destroyed. In detail, the manager stores the connection and the component reference associated with the connection in the

connection pool, monitors the validity of the connection and frees resources associated with the connection when the connection is invalid or will be destroyed.

The common application interface provides an entry for the configuration tools. It allows different implementations of configuration tools to communicate with the connector and configure/re-configure the connector. For example, VCon has been developed to support design time configuration, meanwhile, in DRE a similar function tool has been developed for run-time support.

The IEC 1131-3 port is an interface into the corresponding IEC 1131-3 program. As in previous states, there are small part programs alongside the main IEC 1131-3 program to facilitate the port functions.

Component ports take charge of communication with components. There are two approaches to create component ports. One statically creates ports based on a stored configuration, which is loaded at the beginning; the other dynamically creates ports by using the configuration. Functionally, it operates the component based on the standard method invocation.

Facility is embedded inside the connector; it does not provide services for outsiders. It provides standard services to aid interpreting protocols between the IEC 113-3 port and the component port, which is invoked based on the configuration of the connector.

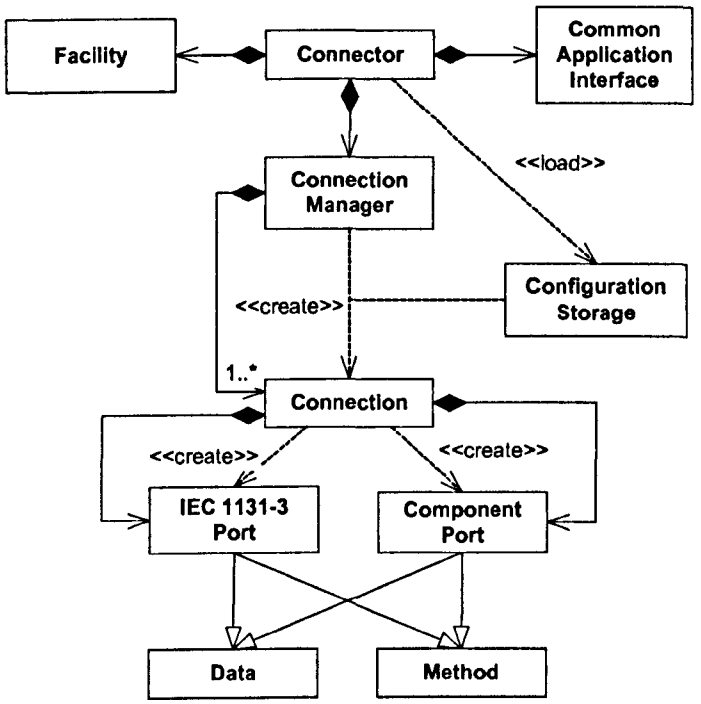

Figure 5.8 Connector Static Structure

5.6.3 Dynamics

The sequential diagram, depending on the scenario, will be used to illustrate the collaborations between the different part participants of the connector. The scenario can be catalogued into design time and run time. While the connector is invoked by the configuring tool in design-time, it is invoked by the IEC 1131-3 program in run-time.

Scenario: Initialisation

Figure 5.9 shows the initialisation process in the design-time, which is specified for VCon. It is the same for other configuration tools. Figure 5.10 shows the run-time initialisation process.

In the design time, the initialisation process is a cascade process, which involves serial creation activities. VCon passes on the selected component ID through the connection manager to the connection. Finally, the connection creates and instantiates the selected component. VCon obtains a reference of the instantiated component as the returned result and the connection manager stores the connection and the associated reference into the connection pool. Based on the reference, VCon browses the component address space; this operation is defined in the VECOM model. VCon creates the component ports via the connection, which connects into the component based on the interface information. After the instantiation of the component port complete, it synchronises values with the connected component through a callback interface, which utilises the publish/subscribe technique. During the design time, the IEC 1131-3 port is not involved.

However, some errors may occur during the process. If a component does not exist, the component port receives the system error and then the error is propagated, making the designer aware. Sometimes a component exists, but its interfaces do not match the requirement. Connection manager needs to detect the interface mismatch. If this happens, connection manager also generates an error.

It is appropriate that VCon and connector are in the same machine, as a result the connector is designed as an in-process component. The composed component is not required to be located locally; it may be located at a remote machine. If the component is located at a remote machine, the component program ID will be added to the remote machine name.

The VCon tool communicates with the connector via the Common Application Interface. The Common Application Interface is well defined in the connector structure. Therefore, as far as configuration tools confirm this interface, the operation process is same.

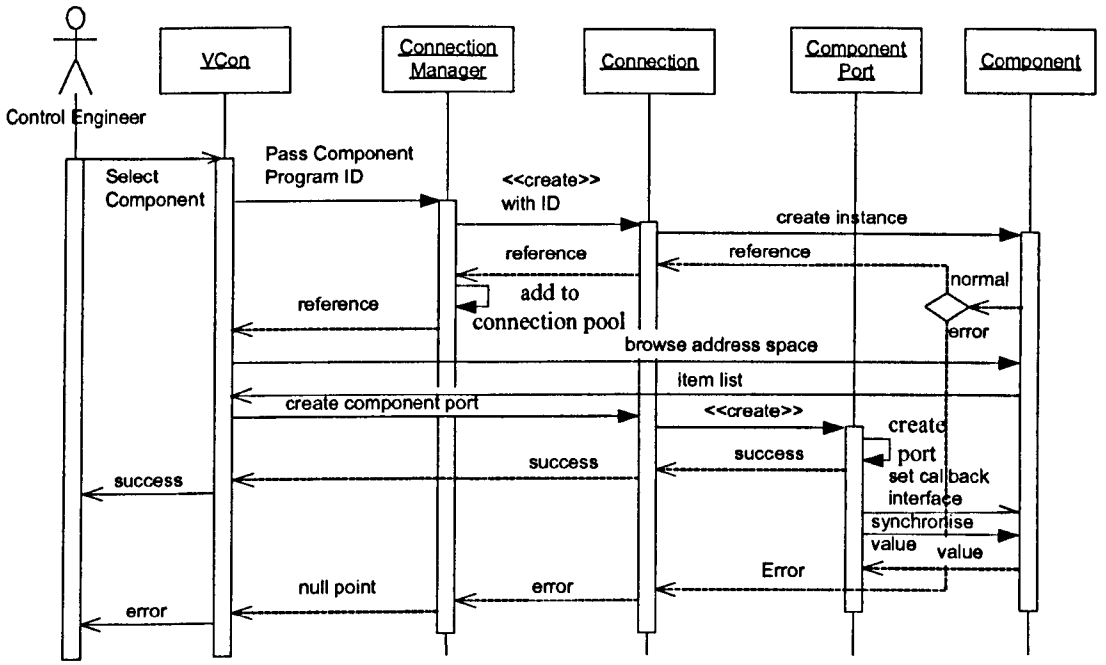

Figure 5.9 Design-time Connector Initialisation

At run time, the initialisation process is similar. But it is assumed that the errors related to the selected component do not occur. Thus, the error routine has been omitted. The component loading process is based on the stored configuration, which is retrieved by the configuration storage function. Similar to the previous one, the IEC 1131-3 program replaces the configuration tool to co-ordinate the process. Finally, the connection instantiates the component, and creates the IEC 1131-3 port and the component port. The creation of the component port is triggered by the component manager based on the storage rather than by the external program. While the component port synchronises values with the component, the IEC 1131-3 port synchronises values with the component port in a data push fashion.

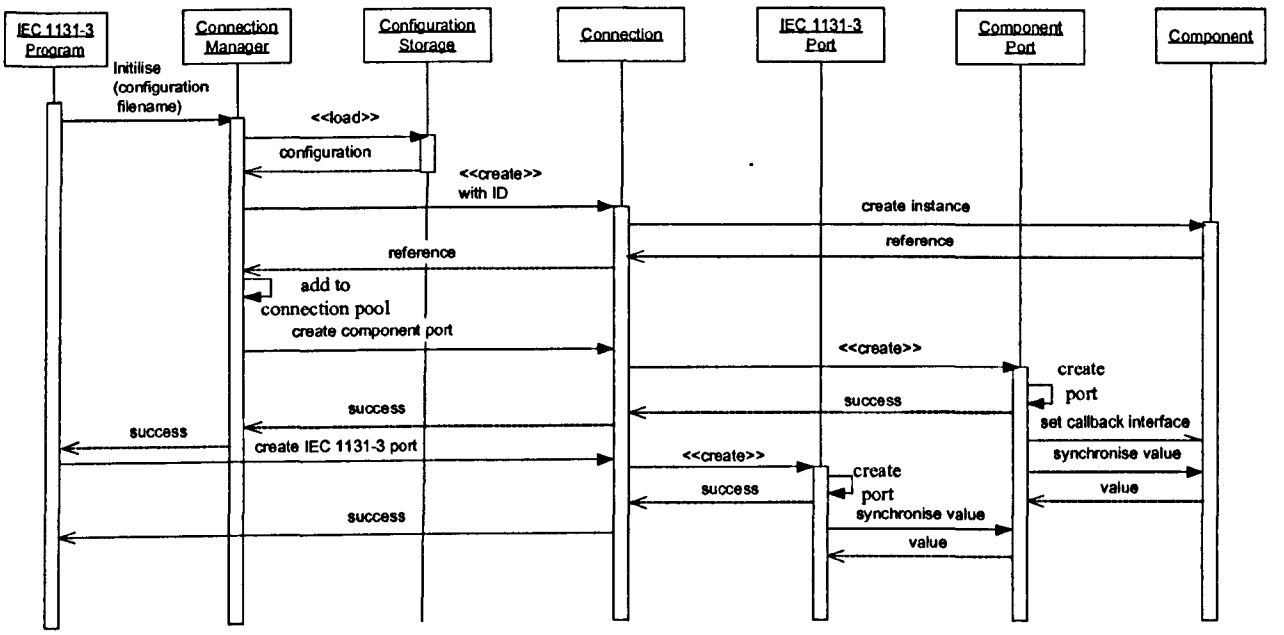

Figure 5.10 Run-time Connector Initialisation

Scenario: Event and asynchronous data update

Figure 5.11 shows the response of the connector to component events, the component asynchronous data update and system event being similar. The routine can be described as the incoming information firstly triggers the component port event and is temporarily stored in the component port. Once the scan cycle time is due, the IEC 1131-3 port is triggered to retrieve the change for the program. Such an approach implies that the IEC 1131-3 program can receive the event and/or data, changing no more than one-cycle time.

The system events that the connector monitors are the specific events, which could affect the connector operation, such as the termination of involved components, system shutdown signal, e.g. Connector does not intend to use the system event mechanism to gather GUI information. Because there are nearly no user interfaces provided by the controller, the run-time support environment deals with it.

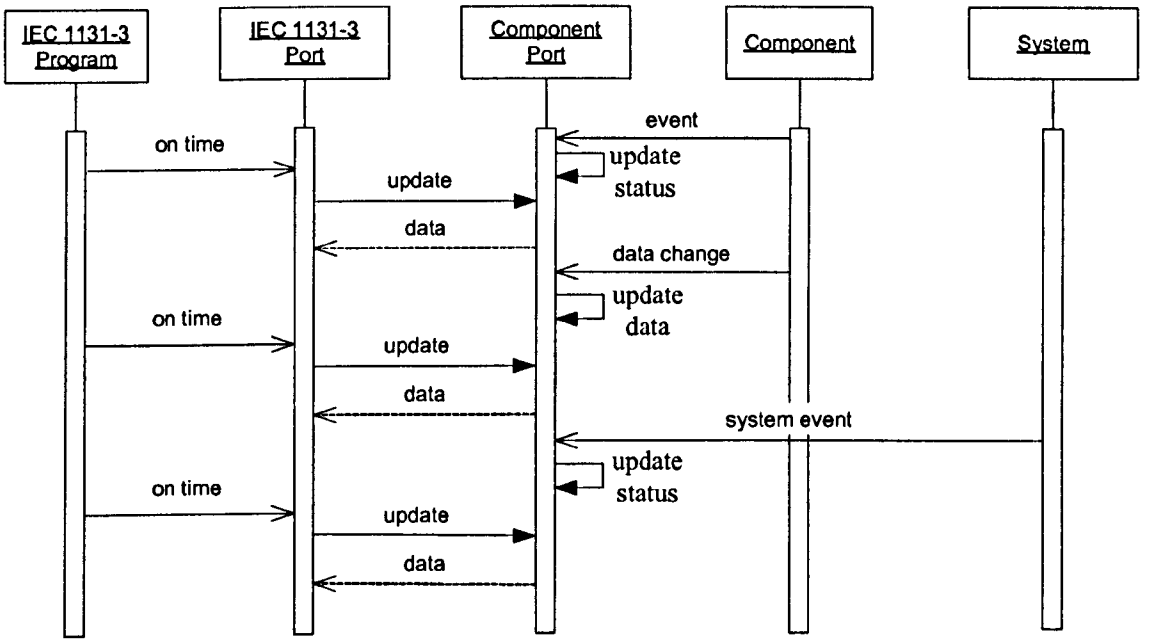

Figure 5.11 Connector Event and Asynchronous Data Update Process

Scenario: Process call and Synchronous update data

As figure 5.12 shows, when a process call and/or synchronous update data occurs, the IEC 1131-3 program and port will be held on waiting status. In fact, the thread control has been passed to the relative component until the call has been returned. This means that any following call will not be carried out until this procedure finishes. Therefore, the process time should be considered.

Here, the process call only happens from program to component. The IEC 1131-3 program is considered as co-ordinator; it sends commands to components that are composed. This is a purely hierarchical structure.

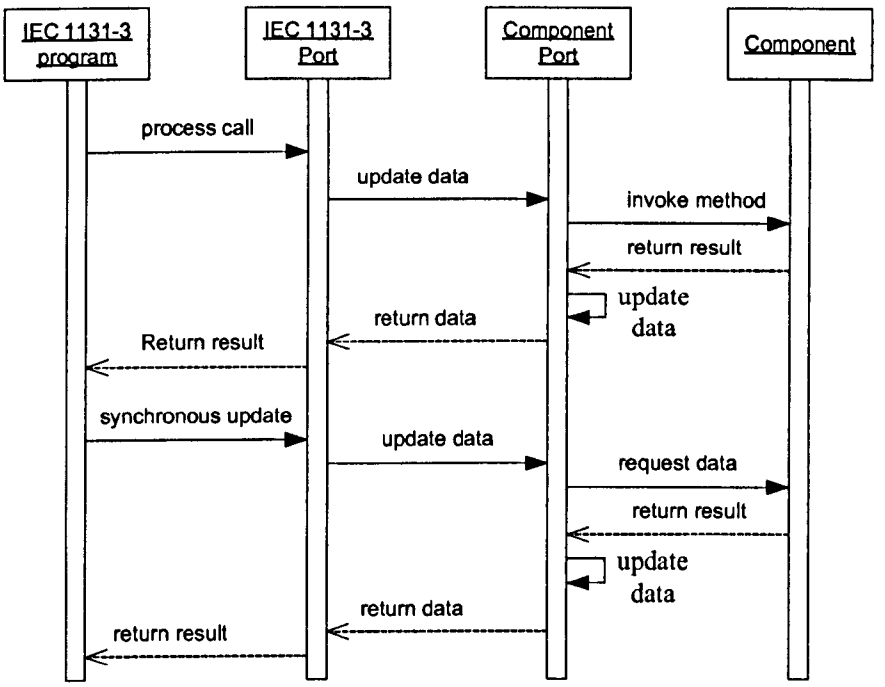

Figure 5.12 Process Call and Synchronous Data Update Process

Scenario: Shutdown

At the shutdown stage, similarly to the initialisation process, the process is in a cascade form. The destroy operation is propagated from the connection manager to the individual port to shutdown relative function elements. It should be noted that before a connection is destroyed, it should wait for the component port to terminate the component and hence free relative resources.

Once the shutdown procedure begins, the IEC 1131-3 program will lose connector control, which means that the connector does not respond to the program any more. So before the IEC 1131-3 program sends the shutdown message to the connector, firstly it needs to finish the whole shutdown operation procedure. For example, the program needs to turn off the actuator before it sends a shutdown command to the connector.

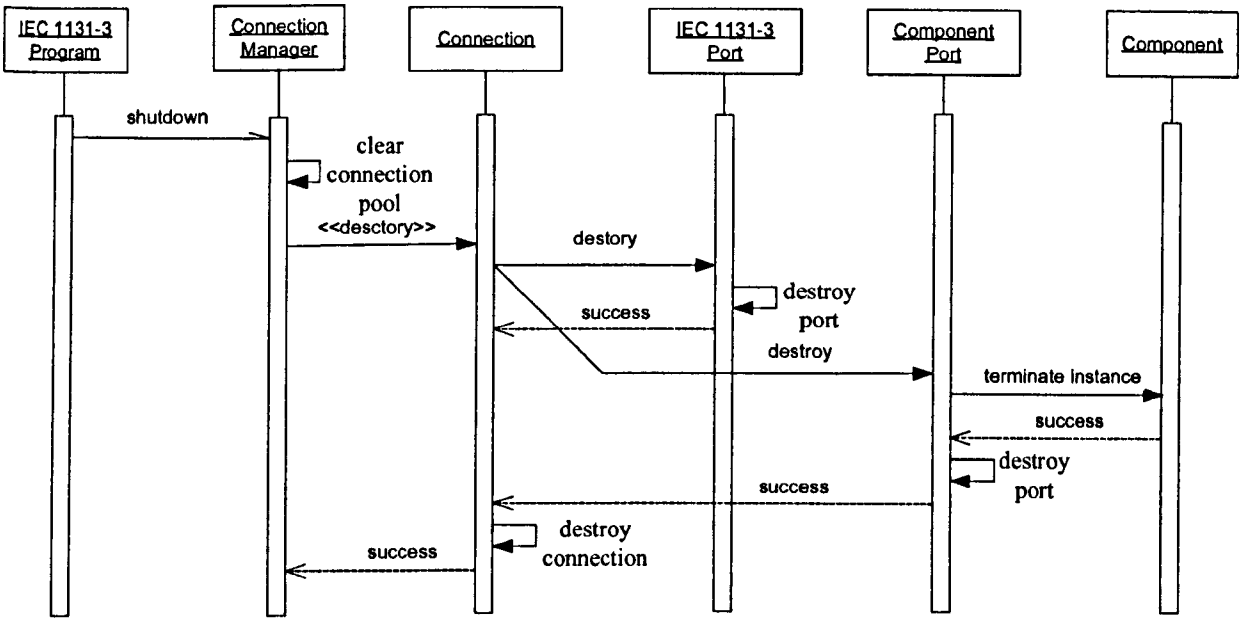

Figure 5.13 Shutdown Process

Scenario: Configuring/Re-configuring connector

To configure the connector, the process could involve the configuration tool (VCon), the connection, the connection manager, the configuration storage, the facility, the IEC 1131 port, and the component port, depending upon the scenario. There are two scenarios related to configuring/re-configuring the connector. One is to create the configuration from scratch, which is simply to create the appropriate IEC 1131-3 ports. Another one is to configure the connector based on the existing configuration. Firstly, VCon retrieves the configuration of IEC 1131-3, and then builds the connection by checking the compatibility of the selected pair ports or inserts a new one.

The incompatibility is caused by the port type and the data type. Since there are two port types in the system, the different types of port cannot connect to each other; this needs to be checked. In addition, the data type could be incompatible. For instance, a 16-bit integer data port might need to connect to a 32-bit integer data port. VCon requests the service from facility to solve the incompatibility. If the service has been found, that means the connection can be established. Otherwise, it is an invalid connection.

After the IEC 1131-3 port has been created, the port will be returned to VCon. So that VCon can create a simulation environment, which allows the control engineer to do a first round test of connector and component performance.

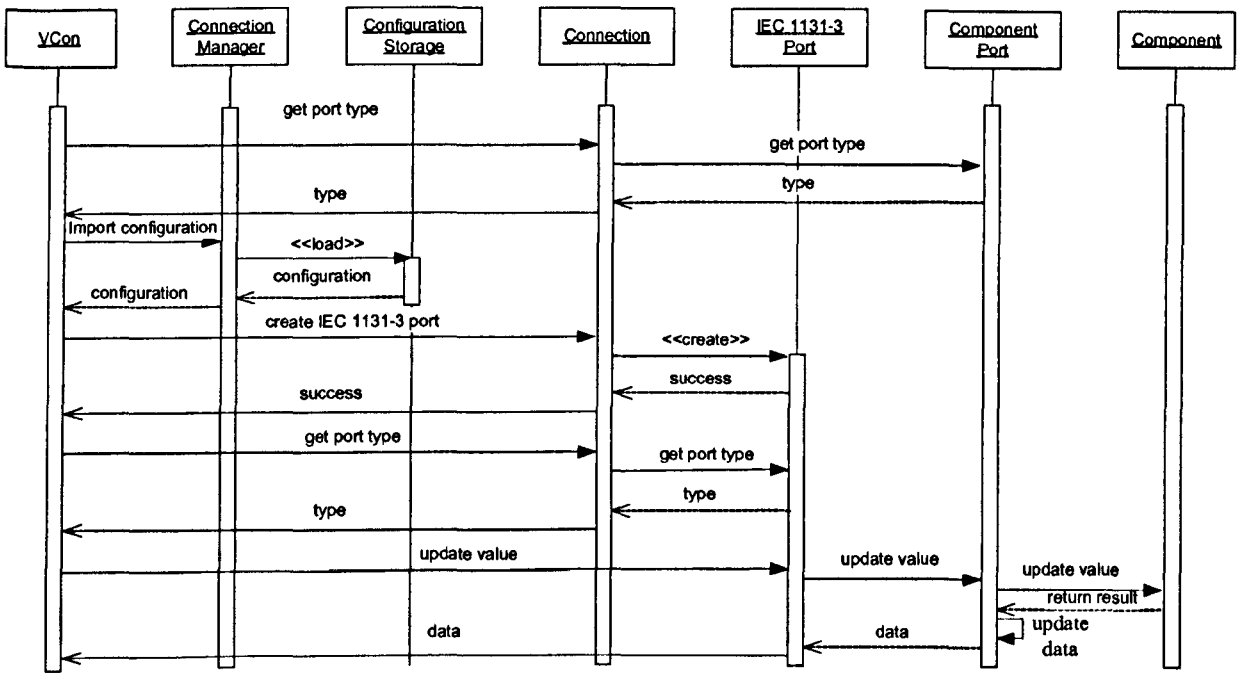

Figure 5.14 Configuring Connector Diagram

5.6.4 Services

According to the above description, the service that the connector provides can be summarised as follows:

- Communication. Communication services support transmission of data between two elements. Data transfer services are a primary building block of the connector. Elements routinely pass messages, exchange data to be processed, and communicate results of computations.
- Co-ordination. Co-ordination services support transfer of control between two elements. Elements interact by passing the thread of execution to each other. Process calls and method invocations are examples of co-ordination services.
- Conversion. These services convert the interaction required by one element from that provided by another. Interaction mismatches are a major hindrance in

composing large systems. The mismatches are caused by incompatible assumptions made by elements about the type, number, frequency, and order of interactions in which they are to engage with other parts. Conversion services allow elements that have not been specifically tailored for each other to establish and conduct interactions. Conversion of data formats is an example of this service.

- Facilitation. Facilitation services mediate and streamline element interactions. There is a need to provide mechanisms for facilitating and optimising their interactions. Mechanisms like load balancing, scheduling services, and concurrency control are required to meet certain extra-functional system requirements and to reduce the coupling between elements.

Every part of a connector provides at least one of these categories. Some parts of a connector provide a 'multi-service'. Formalising part service helps in connector development.

5.6.5 Connector Implementation

Which comes from AlterSys Inc., the ISaGRAF has been adopted to become an IEC-1131-3 program environment and fully supports IEC-1131-3 and the extra Flow Chart Diagram. It uses dynamic linked library (DLL) to implement C Function calls. According to the above factors, the connector is implemented in the form of DLL; it combines with a C Function call to become part of the ISaGRAF kernel. When the ISaGRAF kernel is initialised, the C Function call library is automatic loaded, which means the connector also is loaded into the kernel. The main thread of the connector will be created by the kernel, and then connector will create the necessary thread for every part.

There are two basic ways of exchanging information with a real-time system. They are buffered and unbuffered communication.

Buffered data may arrive at any time. When used in hard real-time systems, upper bounds on the number of produced/consumed messages must be determined to enable guarantee temporal properties.

Unbuffered data is normally accessed through shared memory, which can always be read and written to. Generally, it is easier to check systems temporal requirements. Furthermore, this style of communication is also the most preferred in controller applications. Hence, interfaces of hard real-time components should be unbuffered.

The connector implementation run-time structure is shown in figure 5.15. The connector implementation uses a hybrid method. Actually, every port can be considered as one message handler, from different message perspectives; this is the buffered method. In the meantime a message is dealt within one IEC 1131-3 cycle-time, if the same message comes in more quickly than one cycle-time, this message is lost. Therefore, this is the unbuffered method.

The IEC 1131-3 port thread is controlled by the IEC-1131-3 program and uses shared memory to communicate with the IEC 1131-3 program. In order to achieve this function, five standard C functions have been introduced, which are BoolAddress, SIntAddress, DIntAddress, RealAddress, and UpdateData. For method ports, the corresponding FB will be constructed. UpdateData is used to trigger the IEC 1131-3 port to exchange information with component ports. The relative codes are automatically generated by VCon. Therefore, designers do not need to consider these detail operations.

Connection management runs on another thread. It listens for the event from the system and then responds to ensure that the component connection is valid. If by accident some components have been shutdown, it will inform the IEC-1131-3 program, in order to prevent further damage.

Overall, the connector is a loose coupling method between the IEC 1131-3 program and the components. It provides flexibility and scalability to the program. If some components are upgraded, the relative connector often does not need to be modified. In addition, it is also loosely coupling with the IEC-1131-3 workbench. The workbench change should only affect the relative C function implementation and code generator, not the connector itself.

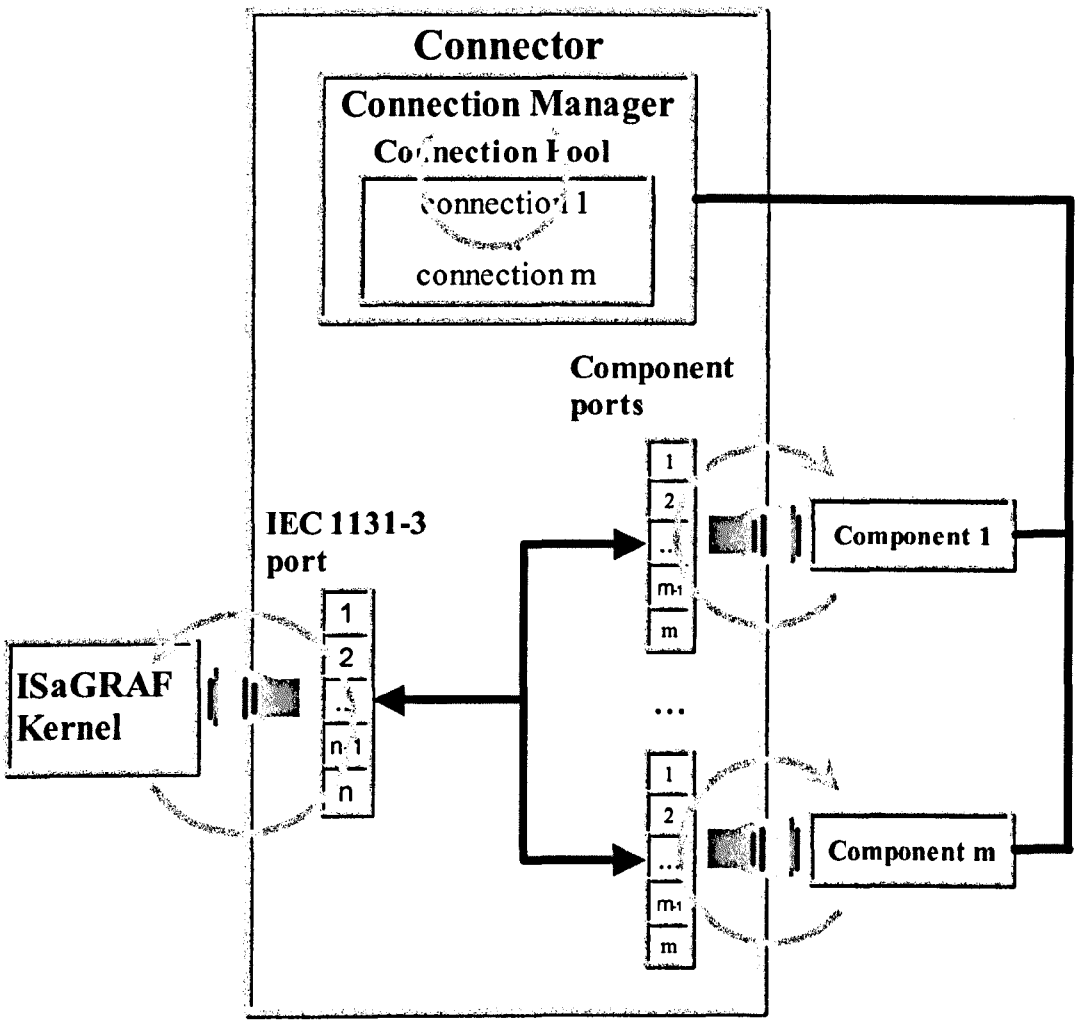

Figure 5.15 Connector Implementation Structure

5.6.6 VCon

VCon stands for VECOM CONfiguring tool. As a part of CLPE, it is designed for supporting connector configuration. The main functions are summarised as follows:

- Retrieving the component ports information. This information is obtained through the VECOM component standard interface and type library file. This information is represented by a tree diagram; the root of the tree is the component name.
- Retrieving IEC-1131-3 ports information. The tree diagram is also used to represent the structure of the information. The ports have been categorised into several groups, so that they are very easy to organise.

- Binding ports. The binding process is achieved by using a binding button to connect the two selections from two columns. The compatibility of the ports will have been checked beforehand.
- Configuring component and connector. Some components need to be configured before they can be used. For example, SDS components need to be configured, in order to check the actual number of hardware SDS sensors and actuators, and also the properties of sensor and actuator need to be verified.
- Testing connector performance. Although the loading of the system is different from the final deployment, it is still worth testing the connector, before the final test. At least the test can verify the hardware connection and configuration correction.
- IEC-1131-3 code generation. This greatly reduces the control engineer's tedious jobs and so avoids human error.

The outline of VCon is shown in Figure 5.16.

In addition, VCon provides a user-friendly interface for the control engineer, allowing the control engineer to quickly connect components to the program. It also tracks the exiting configuration and also allows the control engineer to create consistent configurations. It is an active participant. When connecting ports together, it helps to detect the incompatibility of ports. After one configuration, it allows the control engineer to do rapid prototyping – creating a simulated program port that can be 'test driven' before program testing. The testing interface is shown in Figure 5.17.

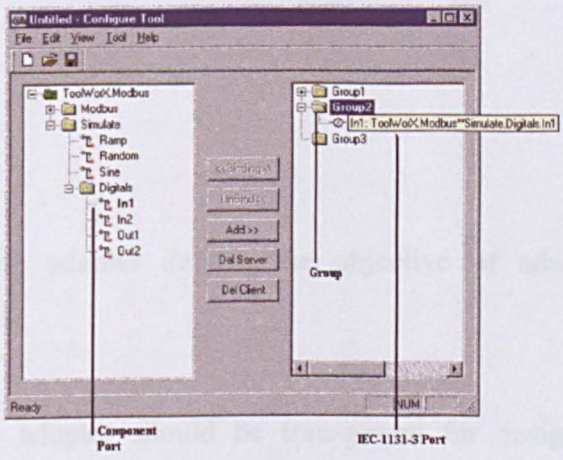

Figure 5.16 VCon Interface

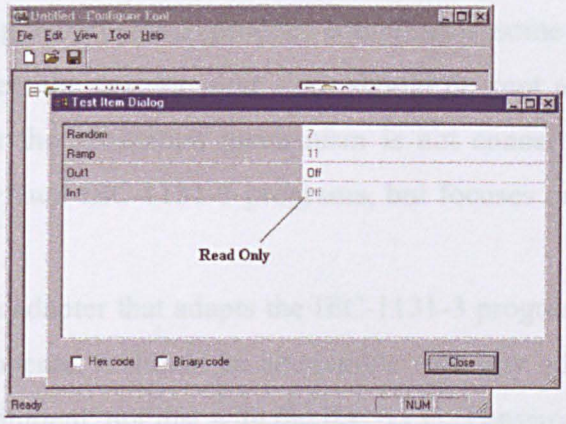

Figure 5.17 Testing Interface

5.7 Adapter

Allowing the IEC 1131-3 program to use components is only halfway to the final target. Historically, an IEC 1131-3 program is not able to generate software components. The traditional way to reuse IEC 1131-3 programs is by using copy-paste source codes. The communication mechanism for outside programs is normally a proprietary method, which depends on the workbench vendor. The adapter method has been proposed to provide an interface-based communication mechanism. It converts the proprietary communication mechanism to a VECOM component. From the integral environment standpoint, it realises recursive component composition operation.

5.7.1 Objectives

Before describing the adapter details, the objective of adapter development is summarised as follows:

- Transparent: The adapter should be transparent for designers and component users. Transparent, in this context, means that both the user of the adapted IEC

1131-3 program and the program itself are not required being aware of the adapter and the adapter does not introduce accidental complexity in the exposed API.

- Black box: During the process to produce components, some development work is required. However, the development work should be kept as simple as possible. This implies that the adaptation mechanism is not concerned with the internal structure of individual IEC-1131-3 programs, but focuses on the communication aspects.
- Composable: The adapter that adapts the IEC-1131-3 program can be composable with other components. It should be compatible with any other component-based developing environment, not just with the IEC-1131-3 environment.
- Configurable: The adapter generally consists of a generic and a specific part. In order that the adapter is useful and reusable, it has to provide sufficient configurability of the specific part.
- Real-time: The adapter does not only facilitate real-time communication as the basic requirement, but also makes it a concern of memory constraint.

5.7.2 Information from IEC 1131-3 program

Although the different IEC 1131-3 workbenches provide the different communication mechanisms for outside, the information provided by and/or to the program generally is the same. The information generally includes:

- Variable. The variable is essential in the IEC 1131-3 program. Here, variable refers to continuously changing data; it maybe changes every program execution cycle time. For example, an analogue sensor value can be treated as variable.
- State. The state is usually used in the SFC transition and ON/OFF actuator. In some cases, it determines the program running direction. Therefore, it can be treated as an event.
- Function/Function Block. In the IEC 1131-3 program, function and function block encapsulate standard processes, which are very important for reusing the program regardless of the implementation. In order to de-couple the adapter and program, the adapter should not directly invoke the function or function block. The invocation function or function block method always depend on the workbench

that the program uses. Adapter directly achieving it need tightly couples with kernel, or even modifies the kernel. Therefore, the function and function block needs to couple with two states, one for invoking it, the other indicating the end of the process.

5.7.3 Adapter Structure

The adapter includes configuration storage, VECOM interface, user-defined interface, port manager, port, kernel communication manager and data cache. The structure can be depicted as in Figure 5.18.

The VECOM interface and the user-defined interface are facilitated by the adapter, which makes the component developed by the IEC 1131-3 program just like any other components.

The port manager creates and organises ports, which connect to the component consumers. It manages the connection requests from the consumers and stores the connection information. When a consumer disconnects, it is in charge of cleaning up the associated memory and destroying the thread. The port is a bridge between a component consumer and the data cache. It sends the information retrieved from data cache to the consumer.

The kernel communication manager creates and manages the data cache, which responds to communication with the kernel. It supervises the communication process and detects errors. In particular, it provides a time trigger for the data cache to gather data. Every consumer request is handled by the data cache.

Configuration storage provides the configuration for element initialisation. In particular, it aids the kernel communication manager to construct the data cache.

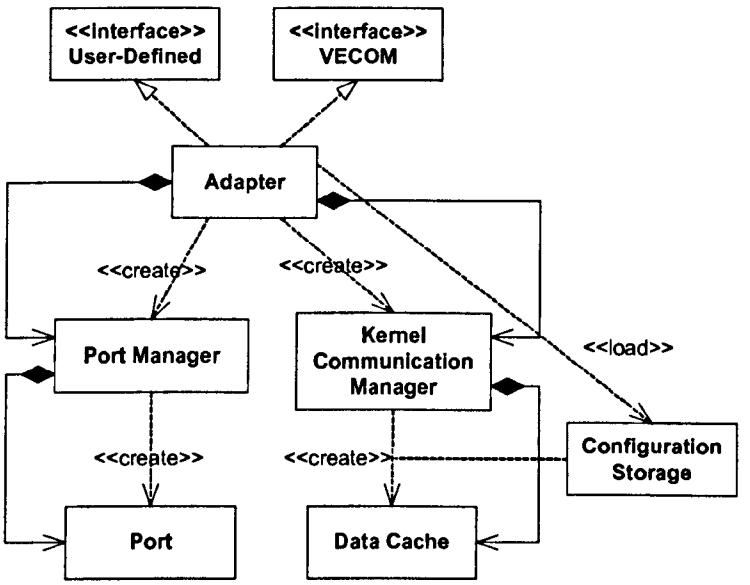

Figure 5.18 Adapter Static Structure

5.7.4 Dynamics

The adapter dynamic behaviour or the collaboration of every participant of the adapter can be described in the sequential diagram. The following scenarios give a clear outline of the adapter in operation.

Scenario: Initialisation

The initialisation sequence is illustrated in figure 5.19. It adopts an eager loading strategy. In this strategy, the kernel communication manager creates the data cache based on the configuration, just after the kernel has been launched, rather than after the creating component request comes. This approach provides a quicker response to the connection request. As a part of the data cache initialisation process, it synchronises its data with the kernel by means of retrieving current data and starts a timer to facilitate the period synchronisation. Benefiting from the eager loading strategy, it can provide the fastest response for a request. When a consumer requires a component via the interface, the port manager creates a port to deal with the request. During the port initialisation, it synchronises its data with the data cache. After the

port creation, the port manager increases the counter and returns a component reference.

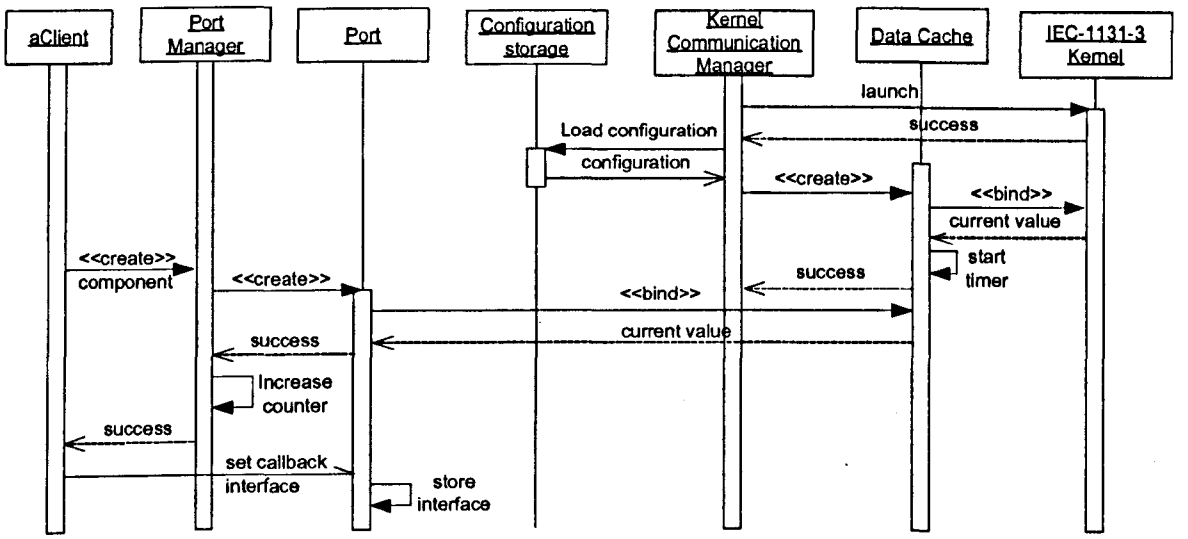

Figure 5.19 Adapter Initialisation

Scenario: Data request and event

The data request and the event notification operation processes are shown in figure 5.20. The data request operation is a synchronised process. It starts with a data request from the consumer, which is handled by the related port via the interface. Depending on the request factor such as deadline and amount, the request is scheduled into the data cache. Data update is trigged by the internal time thread; the update rate is determined by the IEC 1131-3 program cycle time. Finally, the data cache retrieves the requested data from the kernel and returns it to the consumer. The rate is no faster than one cycle time. But, it is not affected by the number of consumers. The event notification process is an asynchronised process. During the IEC 1131-3 program cycle time, the data cache synchronises data with the kernel once triggered by the timer. By comparing the new data with the previous data, the data cache can detect the change and eventually the connected consumer is informed of the change.

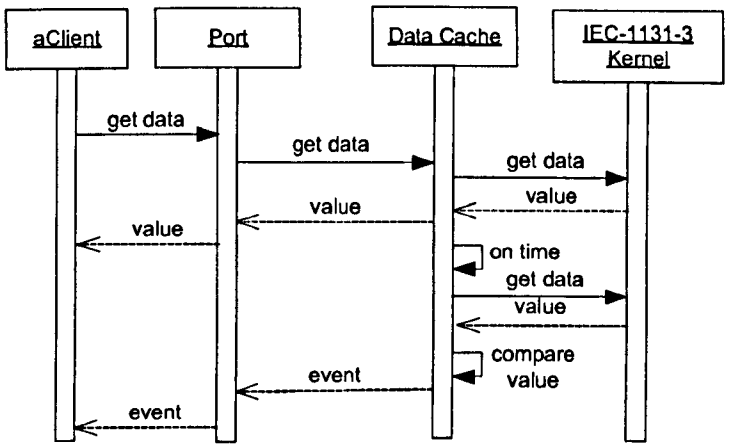

Figure 5.20 Data Request and Event Sequence

Scenario: Method

As figure 5.21 illustrates, the port translates a method call into a data structure, and sends it to the data cache. The data cache must make sure that the parameters of the internal function have been set to the function before the function is invoked. The invoking function is used a flag to achieve. When the function is fired, the flag needs to be set; when the function is ended, it needs to re-set, so that the result can be retrieved. This requires the IEC 1131-3 program to confirm this requirement. The checking finish-state action may occur over several cycle-times; it depends on the function implementation. As a matter of fact, it is a synchronous process, which means the component will take over control until the process finishes. After the function finishes processing, the control returns the caller. The asynchronous process call is not designed into the adapter. For example, the call back function is not supported in this adapter mechanism. It can be achieved by using the event-informing mechanism.

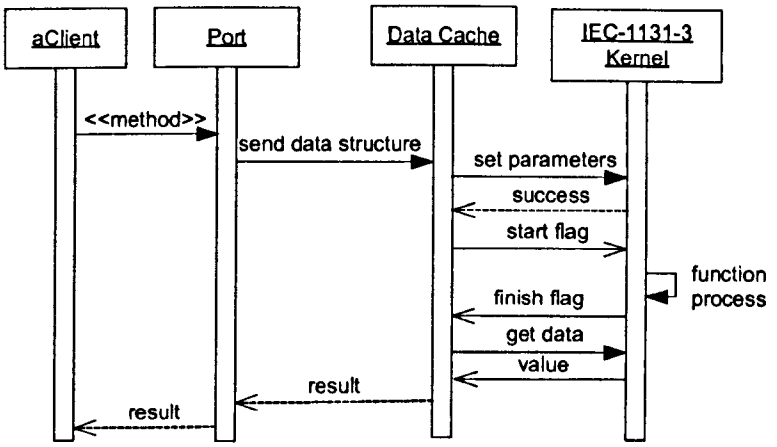

Figure 5.21 Adapter Process Call Sequence

Scenario: Shutdown

The shutdown process is shown in Figure 5.22. Corresponding to the eager loading strategy, it adopts the eager unloading strategy, which can provide a small footprint for the overall system. Once a consumer issues a destroy command, the port manager destroys the relative port and clears the associated resource. Then it will check whether the counter reaches zero or not. When the counter is equal to zero, the port manager informs the kernel communication manager to start the final destroying process. The data cache is firstly destroyed, which includes stopping the internal timer. Then the kernel manager terminates the IEC 1131-3 execution kernel. After that, the kernel manager clears the allocated resource and terminates itself. Finally, the port manager completes the whole shutdown process.

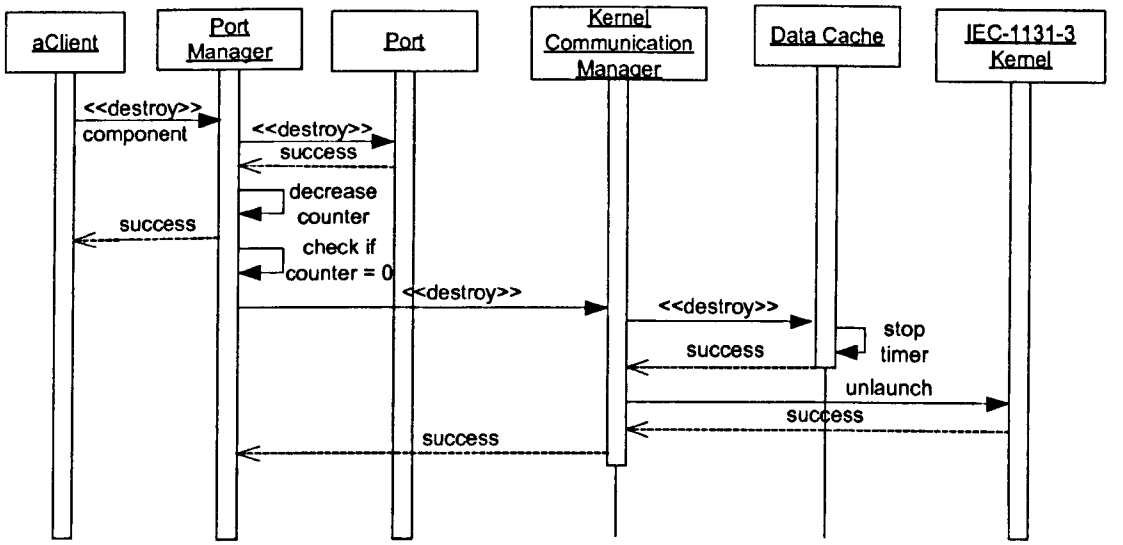

Figure 5.22 Adapter Shutdown Sequence

5.7.5 Adapter Implementation

The implementation structure of the adapter can be depicted as in figure 5.23. Here, the implementation relies on the basic platform execution kernel, which is the ISaGRAF kernel. The main association with the kernel is the use of the communication mechanism. The ISaGRAF kernel defines a communication mechanism named IXL (ISaGRAF Exchange Layer) in the form of a set of API for exchanging information, which includes synchronous communication and asynchronous communication. Within the adapter, the data cache wraps the IXL interface and responses to communication with the kernel. In addition, it uses a timer to synchronise its data with the kernel. The data cache only supports four simple data types that are defined by ISaGRAF, which are Boolean, 8bit integer, 32bit integer, and 32bit-float point. There is no other data type available, because of a platform constraint. Ports enable the implemented interfaces and communicate with the consumers. In this way, the port mechanism supports multiple consumers, which are organised by the port manager. The consumers are enabled for communication with the kernel through the communication channels between the port and the data cache, which achieves the design objective.

In fact, an adapter needs to be developed, compiled and deployed as a part of a component. The development process should not be much different from a standard

COM component development process. Thus, a template is provided to ease and promote the development process. The template uses C++ as the implementation language and the development work is carried out in Microsoft Visual C++ environment, which is a standard choice. Inside the template, the issues related to the implementation such as multiple thread co-ordination have been well considered. Most of the works, which includes generic and specific parts, can be done by means of configuring the template.

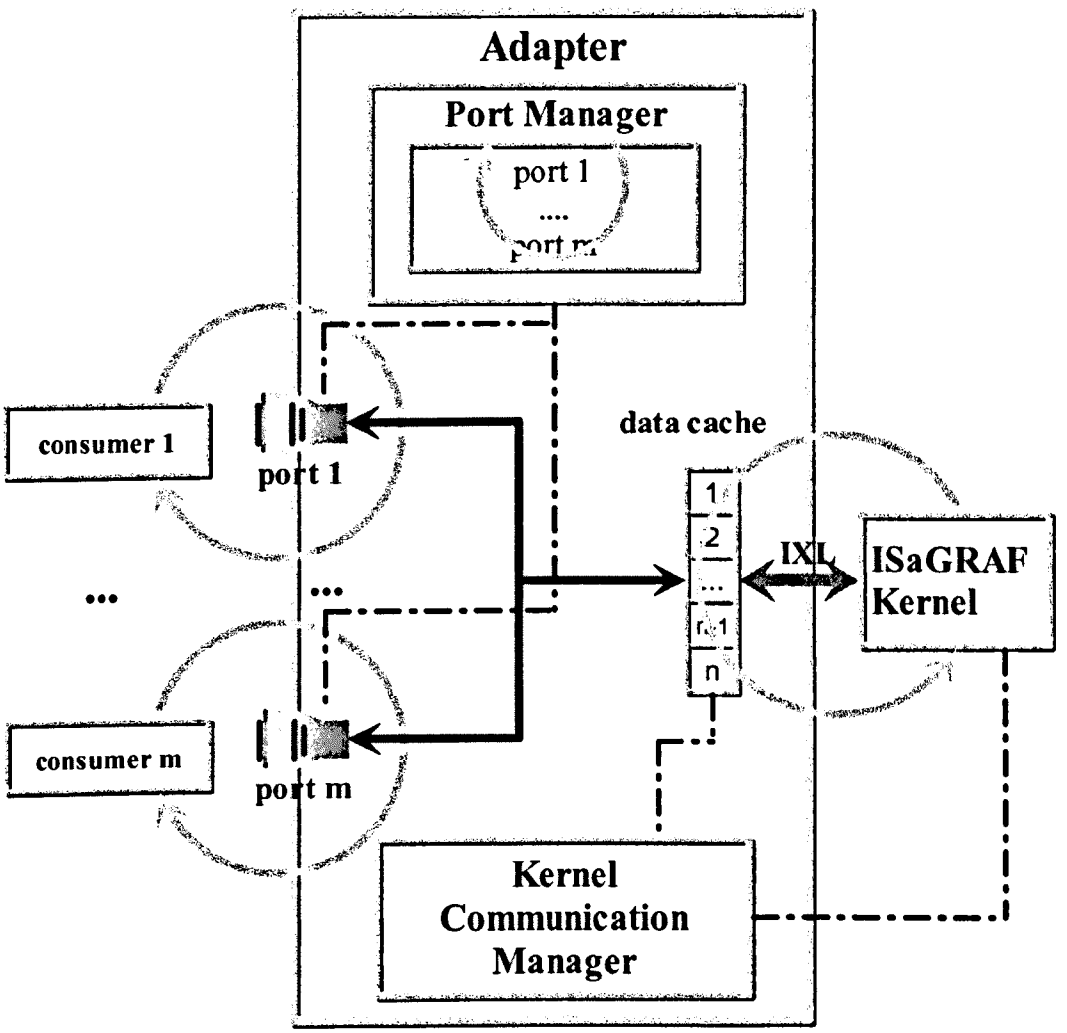

Figure 5.23 Adapter Implementation Structure

5.7.6 CBuilder

In order to assist the development, a supporting tool named CBuilder standing for Component Builder has been developed. It simplifies the development work by

providing a user-friendly interface. The user-interface of CBuilder is shown in Figure 5.24. After the designer selects the IEC 1131-3 program that needs to be converted, CBuilder automatically retrieves the program information, which includes variable name and function name. The designer just needs to pick up the name and drop to the output column. Designer also needs to decide the writable permit of the variables. After the completion of the configuration, CBuilder generates the propriety source code for the component. Through the compiling process, the construction of a component is completed.

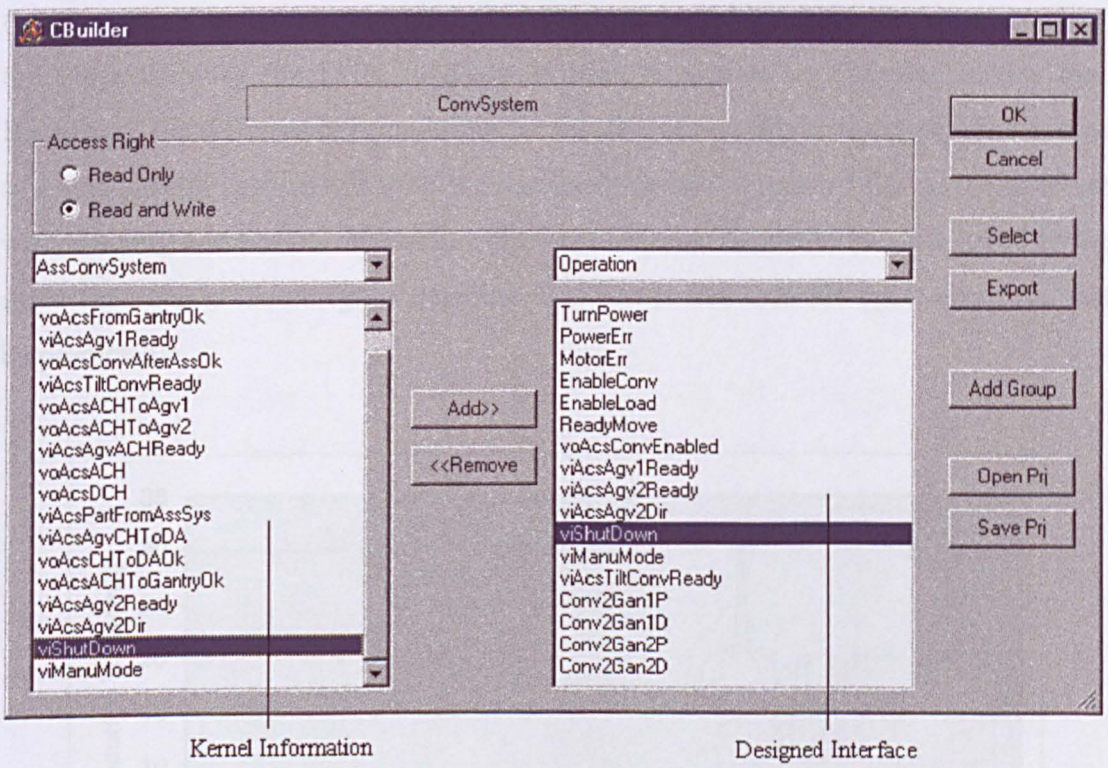

Figure 5.24 CBuilder Interface

5.8 Evaluation

The component provided by the above approach is supposed to provide the real-time behaviour, in other words, the execution time can be predicted. Although the real-time performance has been explained in that it is based on the connector and adapter working mechanisms, it is worth testing in full work-loading condition. The real-time operations can be separated into two kinds, one is 'Read operation' where the

components receive the signal and pass it on to the IEC 1131-3 program; the other is 'Write operation' when the program sends a command or data to the components. In order to achieve the test, a little modification has been made to the connector and adapter by combining the time stamp with data. For the read operation, when the adapter receives some data that needs to be passed on, the adapter records down the time; at the time that the data is available by the program, the connector records down the time. By comparing the two time-stamps, a time cost can be obtained. For the write operation, the time stamp working mechanism is the same; the only difference is that the connector and adapter switch roles. Figure 5.25 shows the absolute time in two domains. One involves he SDS field-bus system, as shown by the red colour line; the other involves the LON field-bus system, as shown by the blue colour line. Because of the different expected time for the different systems, the cycle time is used as a standard for comparison with the results, as shown in figure 5.26. At the different work loading conditions, the passing time is different. Nevertheless, compared to the cycle-time that is the designed deadline, the time is less than the cycle-time, and can be predictable.

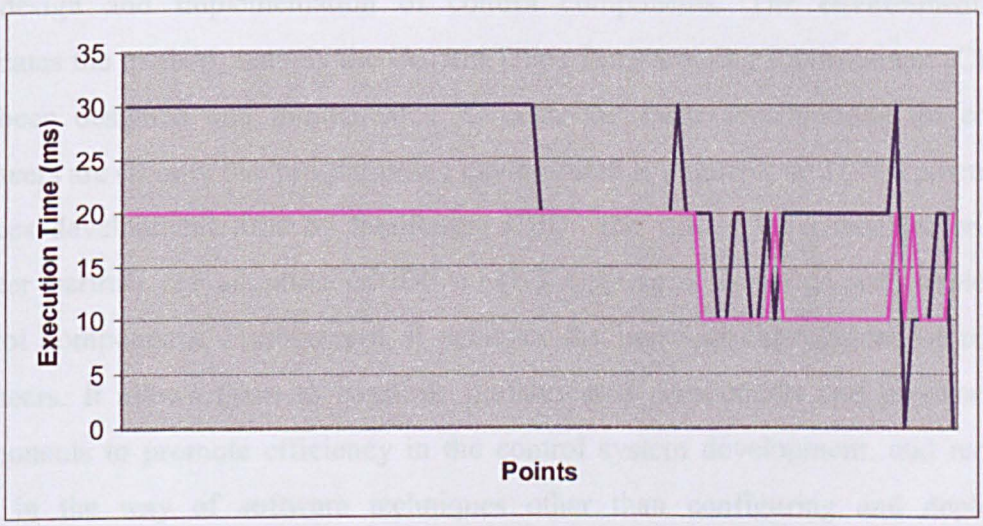

Figure 5.25 The Absolute Time Graph

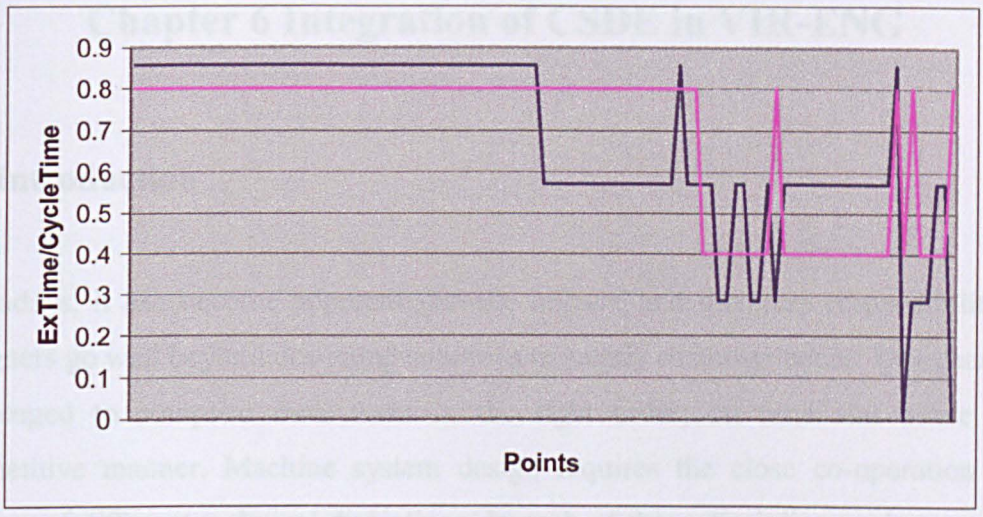

Figure 5.26 The Relative Time Graph

5.9 Summary

To conclude, this chapter has dealt with the methods, which enable IEC 1131-3 for the design and implementation of control components. The environment that facilitates the method, namely the Control Logic Programming Environment (CLPE), has been designed and implemented. Overall, the main contributions for control engineers are (i) only one programming environment is required; and (ii) it potentially reduces development time by facilitating CBD. The use of the connector and the adapter permits the adoption of IEC 1131-3 languages to design and implement control components. Furthermore, it achieves the practical expectations of control engineers. It allows them to consume prefabricated components and produce new components to promote efficiency in the control system development, and requires little in the way of software techniques other than configuring and deploying elements. Associated with the two mechanisms, two tools named VCon and CBuilder respectively have been developed, which are combined with the ISaGRAF workbench to form the whole CLPE. This environment is further discussed in chapter 7 using a complex test case.

Chapter 6 Integration of CSDE in VIR-ENG

6.1 Introduction

Nowadays, it has become apparent that the impact, and therefore responsibility, of designers go well beyond designing solutions to satisfy customer needs. Designers are challenged to complete their tasks in the right order, on time and hence in a competitive manner. Machine system design requires the close co-operation of a number of different technical disciplines. In each of these disciplines, a large variety of software tools are available for analysis and design, each of which provides a specific platform for proper operation. Achieving co-operative design requires structuring system development into a more effective whole and integrating the software tools involved. This helps designers to unify their different perceptions and experiences in their design decision processes. In addition, such practices positively enables unpredicted requirement change rather than inhibiting it.

The above objective can be achieved by "integration", which refers to the composition of the VIR-ENG environments, broadly including MMDE (Modular Machine Design Environment), DCSE (Distributed Control System Environment) and IIS (Information Infrastructure and Integration Services). In particular, IIS provides workflow capabilities and groupware facilities for the integration. While workflow capabilities facilitate information exchange across tools, groupware facilities enable groups of workers to collaborate for some purpose or task.

In the context of this research work, the integration of the Control System Design Environment (CSDE) focuses on two closely interrelated aspects, namely workflow and mechanism. The workflow mainly concerns the external relationship with CSDE, including (i) when should or could it be combined with other tools; (ii) what is the proper tool to enable design process; and (iii) what are the inputs to and outputs from the tools. The mechanism mainly focuses on the internal structure (i) to enable the effective use of the tools, and (ii) to develop and integrate the tools.

Based on the VIR-ENG design process and the component-based machine system design and development philosophy, the CSDE interacts with MMDE, and DRE. Two interactions between MMDE and CSDE are:

- Control requirement and design transfer;
- Control logic program transfer.

For DRE, there are also two interactions with CSDE:

- DRE analysis and design support;
- Runtime support system connecting to the control system.

Before the discussion of the CSDE contribution to integration, IIS is firstly described to outline the overall facilities and mechanisms of integration. The following content is organised in the order of the interaction points.

6.2 IIS

IIS has been developed to provide information infrastructure and integration services to support a highly integrated design, simulation, and distributed control environment for constructing machine systems. The IIS – integration platform as shown in Figure 6.1 consists of three major components:

- IIS - Component Library
- IIS - Component Bus
- IIS - Component Manager

The IIS – Component Library can be considered as a system-wide database that provides persistent storage and management of components within VIR-ENG. The IIS – Component Bus provides the essential information services, which enable components stored in IIS – Component Library to be accessible from a wide range of applications and most web browsers. In fact, it allows the collaboration between design tools to achieve a heterogeneous fashion. The IIS – Component Manager is a project management facility with two main features: (i) a graphical user interface for the IIS – Component Library to enable access to the components and services; (ii) act

as middleware to enable applications to synchronise information with the information storage.

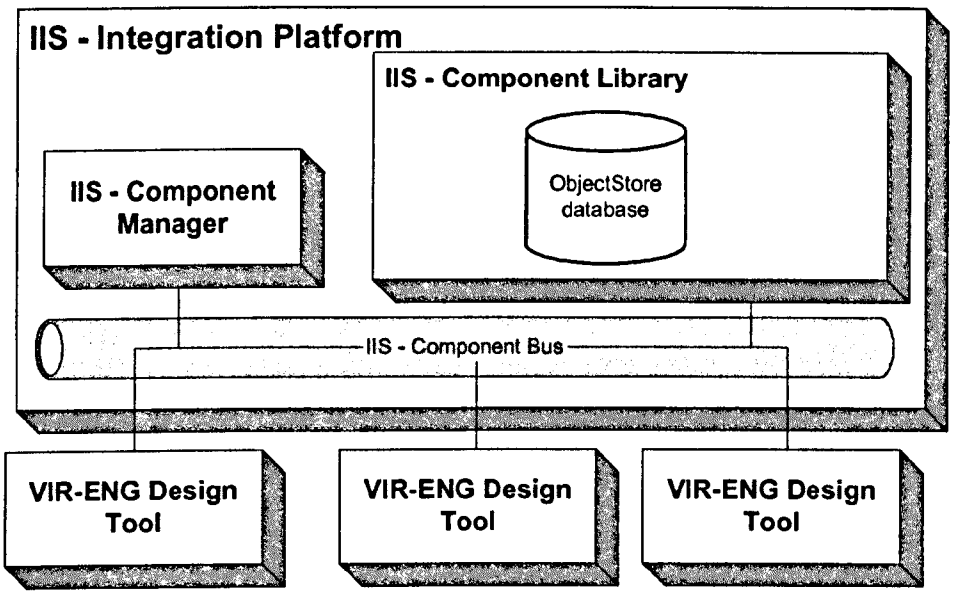

Figure 6.1 The IIS Structure

One of the key benefits of IIS is that it enables workflow via the Internet or Intranets to allow collection and dissemination of information among system designers in a transparent manner. Furthermore, it provides a means for managing distributed information systems, by propagating information between design nodes and by synchronising the activities involved in the design processes, which is regarded as the implementation of groupware. As the considerations of CSDE, it shields users and developers of the design tools from the communication complexity and provides persistent storage for various VIR-ENG components via a system-wide component library.

6.3 Integration with MMDE

6.3.1 Control Requirement and Design

The first interaction between MMDE and CSDE integration occurs at the control architecture design stage. The control requirement specification derived from the

result of mechanical design and simulation is transferred to CSDE to start the control system conceptual design process. In return, the designed control architecture specifications are fed back to the MMDE for both verification and programming work in subsequent stages. As CSDE can be further divided into the Control Architecture Design Environment (CADE) and Control Logic Programming Environment (CLPE), the relevant design activities take place within CADE. As the illustration in Figure 6.2 shown, the major output from CADE is the component interface specifications, which must be compliant by all VIR-ENG environments during the subsequent control system development process. Such specifications, represented using UML-based diagrams and data tables, include the following information:

- A list of control components, and their collaboration – this is the result of decomposing the system by using Component Responsibility and Collaboration (CRC) methodologies.
- Specifications of the component interfaces, which define the responsibilities of the components and define component operation methods.
- Event logic has been represented in the component dynamic collaborations that are depicted in sequential diagrams. Most of them can directly translate to SFCs, which outline high-level states and sequences, and describe logic relationships between the components. While the concrete internal logic and code of the components are left to MMDE and CLPE during the control logic programming stage. Together with the interface specifications, these SFCs provide code skeletons for the subsequent programming work.

Tools within CADE are deliberately designed to support component-based control system design. In particular, it provides a graphical editor for control engineers through construct task decomposition diagrams to formulate information passed from MMDE and unify the common understanding between two domains. In addition, its inherent capabilities to support the CRC method and construct UML diagrams enable capturing different views of the conceptual design. As the part output, report facility extracts diagrammatic interfaces description into table format for confirming control engineer work tradition in Word/Excel format. Chapter 4 provides detailed descriptions of the CADE and the CRC method.

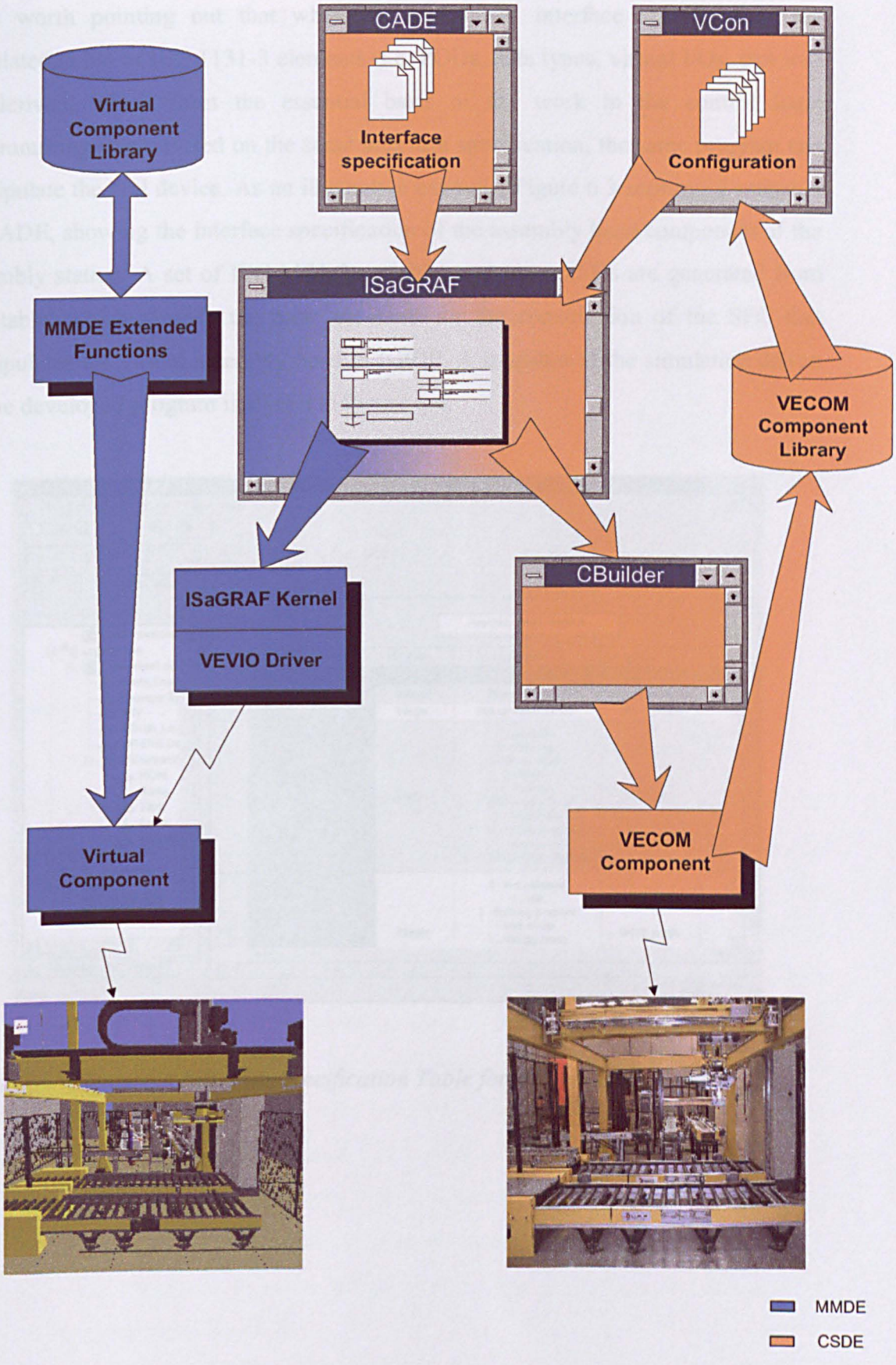

Figure 6.2 Tools Integration between MMDE and CSDE

It is worth pointing out that when the component interface specifications are stipulated, a list of IEC 1131-3 elements (i.e. POUs, data types, virtual I/Os, etc) will be derived, which form the essential basis of the work in the control logic programming stage. Based on the same interface specification, the same program can manipulate the real device. As an illustrative example, Figure 6.3 captures a snapshot of CADE, showing the interface specification of the assembly head component of the assembly station. A set of IEC 1131-3 variables and virtual I/Os are generated from this table, which provide the basic elements for the construction of the SFC that manipulates the virtual assembly head in IGRIP. A snapshot of the simulation driven by the developed program is shown in Figure 6.4.

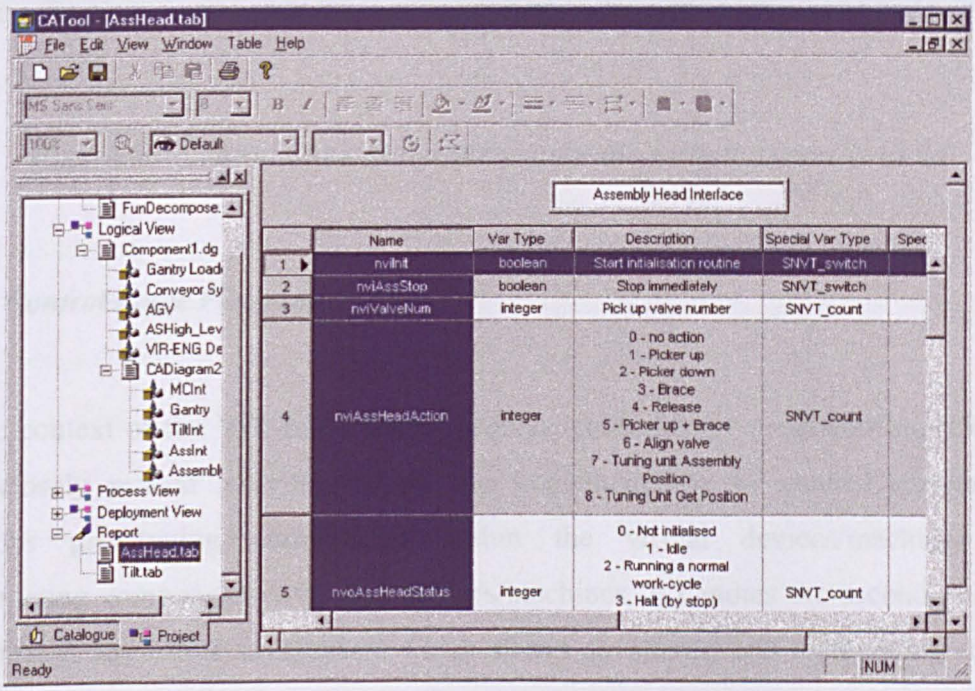

Figure 6.3 Interface Specification Table for the Assembly Head

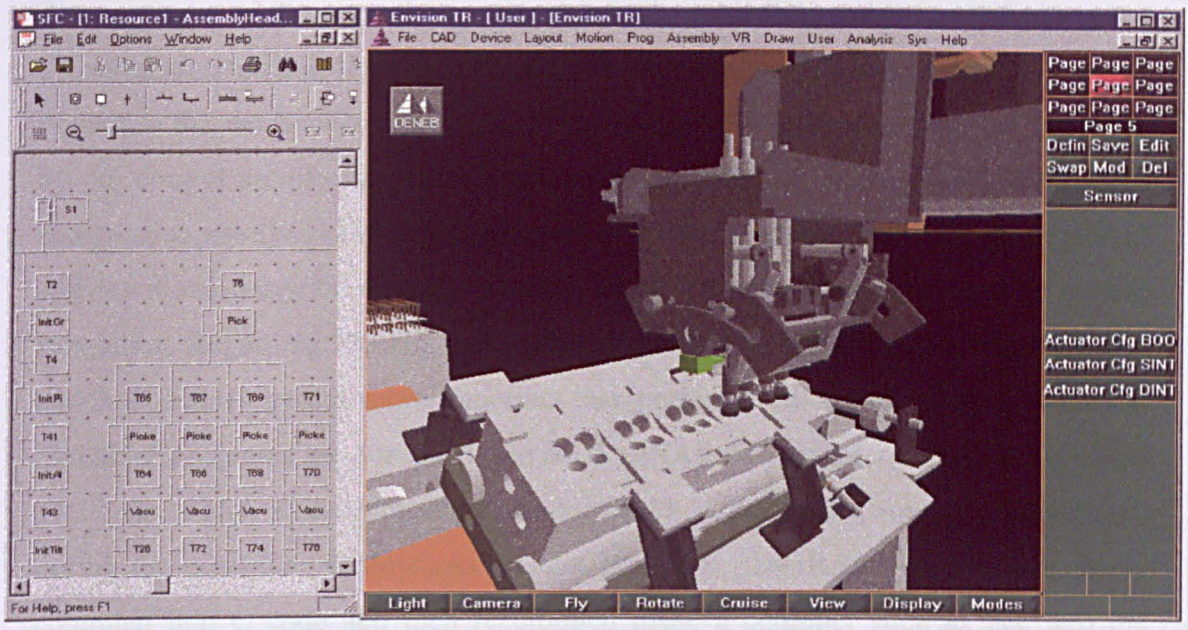

Figure 6.4 Control Logic Programming for the Virtual Assembly Head

6.3.2 Control Logic Program

In the context of the VIR-ENG design process, control logic programming refers to two closely related activities where the implementation of control components includes prototyping components within the virtual devices/machines and constructing components with real devices/machines. It denotes the second point of the MMDE and CSDE integration, which allows developing and maintaining control components in an efficient way. Initially, control components built in the virtual environment can be forwarded to the real environment driving real machines or devices. Later, control components used in the real environment can be rolled back to the virtual environment for re-designing or testing with virtual machines or devices. Integration here is realised in the sense that: 1) MMDE and CLPE share the use of the same IEC 1131-3 languages and the same programming environment, i.e. ISaGRAF supplied by CJ International; 2) the programming work is based on the same set of control architecture specifications.

The prototype of the control component is built in MMDE and it is verified against the simulation models developed in Quest and IGRIP. The left hand side of Figure 6.2 illustrates the mechanisms involved. With reference to the component list produced by the control architecture specifications and utilising or reusing the control logic components, the IEC 1131-3 logic programs developed using ISaGRAF are compiled and downloaded into the ISaGRAF target, which executes the designs to carry out the simulation. Regarding MMDE as a testing environment, quickly compiling the codes and then running tests is more important than performance. Based on this hypothesis, ISaGRAF programs are compiled into an ISaGRAF-defined intermediate code called TIC code, which is interpreted by the ISaGRAF target during runtime. In contrast, during the controller deployment process in CSDE, ISaGRAF programs are firstly compiled into C code and then recompiled using a compiler such as gcc into the native machine code, in order to leverage the performance. CBuilder is the tool that supports this process.

Communication between the ISaGRAF target and the simulation software is established through the in-house developed VEVIO module. This module enables synchronised data sharing between more than two applications. A customised ISaGRAF device driver called the VEVIO driver has been developed for the ISaGRAF target. Simulation entities respond to the control signal from the ISaGRAF target and produce the corresponding state change and graphical update. Their behaviours are determined by the virtual components developed/generated using the native programming languages of the simulation software. Development of the virtual component controllers is supported by the "MMDE extended functions" module, which is a collection of extended functions developed atop the simulation packages. The MMDE extended functions are packaged in the form of Dynamic-link Libraries (DLLs) so that they can be tightly integrated to the simulation applications (i.e. Quest and IGRIP).

The right hand side of Figure 6.2 illustrates the mechanism within CLPE and the environment in which control logic programs for the real devices/machines take place. This environment is composed of three main modules: 1) VCon (VECOM Configure), the tool that aids the control engineer to switch from the virtual component to VECOM components; 2) the ISaGRAF programming environment, which is the

common programming environment shared by MMDE and CSDE; 3) CBuilder, which supports the conversion from the IEC-1131-3 code to VECOM component(s).

The VECOM model is the essential element of DCSE, in terms of seamless integration of CDE, CSDE and DRE. The CSDE tool-set provides full support for the VECOM component development, which includes re-using the off-the-shelf VECOM components and producing new VECOM components. VCon discovers and configures the VECOM component deployed locally or remotely. Based on the VECOM model, a VECOM component produced by CSDE can be theoretically used by any other language or environment. For example, DRE uses VECOM components in a Visual Basic environment. On the other hand, VECOM components developed by the other environment can be used in CSDE.

The programming control logic uses the ISaGRAF as the basic platform. Fortunately, the control architecture defines the common specifications for MMDE and DCSE (including DRE). Ideally, control programs that drive the virtual models should not have any difficulty in manipulating real machines/devices with only minimal modification to map the interfaces from virtual ones to real ones. Since the virtual environment is an ideal model of the system, it can be easily understood that in practice not all physical aspects and/or the real complexities have been considered. Certain extra modifications are needed when the control engineer is working in CLPE, which include:

- Hardware-specific operation. For example, most hardware components require certain initialisation routines to be executed before operation. Because it is not necessary to be considered in MMDE, the initialisation logic should be considered in CLPE.
- Exception handling. Some exceptions, like sensor failure, can be considered in the virtual environment in order to test the control logic of handling such errors. Nevertheless, there exist a lot of other kinds of failure that are impossible to establish in the virtual environment. Extra exception handling logic has to be added on.

It should be emphasised that the programming involved in CLPE is regarded as add-on extra functions, without altering the original logic verified in the virtual environment.

It is arguable that the component-based development approach offers a more convenient way to achieve such transference. However, the IEC 1131-3 standard and languages do not inherently support the component-based programming paradigm. CLPE achieves the component-based development by means of combining IEC 1131-3 programs with a VECOM component framework named 'adapter'. By using Visual C++ compile, IEC 1131-3 programs are finally converted to VECOM components in the same way as other components developed by other methods. CBuilder is the tool supporting such practice. A full description of CBuilder and the underlying mechanism has been given in chapter 5.

From the control engineer's perspective, even though the activities take place in two environments, the main work is carried out in the IEC 1131-3 basic programming environment known as ISaGRAF. Because of this, the control engineer can easily achieve the whole control system development without noticing the shift between the two environments. The control engineer can smoothly switch from one environment to another at any stage and practically achieve the multidisciplinary co-design fashion. Compared to the traditional development process, developing the control system in the virtual world has a big advantage in terms of reducing development time and early detection of design flaws. In the virtual world, the enhanced graphical representation and visualisation facilitate the discovery of potential defects (e.g. collisions between parts). Also, the majority of the control logic can be comprehensively tested without causing hazards.

6.4 Integration with DRE

6.4.1 DRE Analysis and Design support

During runtime support requirements analysis, information related to the machine system is obtained from the control architecture to help formulate the runtime support

requirements. In return, these requirements are fed back into the control architecture, often resulting in changes to the component interfaces and information updates to the control architecture. The process ends when designers fulfil the runtime support requirements, and the control architecture has taken into account these requirements with updates to the component interfaces and other relevant information such as task sequences, exception handling, etc.

For control architecture design, it is required to provide enough information to support RTS, which includes monitoring, configuration, alarm handling, diagnostics, and maintenance. Control architecture design considers runtime support requirements during definition of the component interface. After control architecture design is completed, runtime support analysis is carried out with reference to the VIR-ENG runtime support reference architecture. This activity uses UML as a modelling language, which can be supported by CADE facilities. Besides using UML for modelling, runtime support design needs some extra notation for design, such as Human Machine Interface (HMI) navigation structure, database operations, and so on. The CADE environment, where specific notations can be added in easily to support other representation requirements, is open for customisation and has facilitated this activity. An example of navigation structure is shown as figure 6.5. In fact, runtime support analysis and design can take place in CADE. It enables runtime support analysis and design to easily obtain information about control architecture and influence the component interface definition, in order to avoid conflicts at various stages.

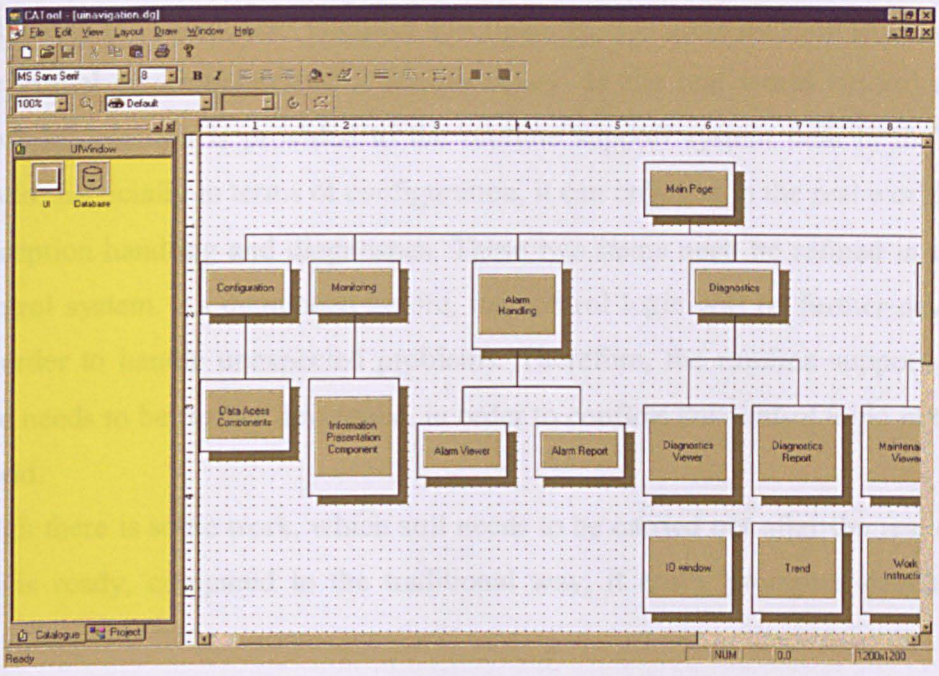

Figure 6.5 An Example of the Navigation Structure

6.4.2 The Connection between Runtime Support and the Control System

Following the VIR-ENG design process, runtime support development is carried out in parallel with the control system development. Before it connects to the real control system, the verification of runtime support design is carried out in the virtual environment. This is a major achievement of VIR-ENG in terms of cutting lead-time. In the traditional development process, this activity always waits for the control system to be ready. Because the early control system development is based on the virtual environment and there will be no major change in control logic, the runtime support system can use it to carry out early verification. This facilitates the design and refinement of the runtime support system before it connects to the real system.

By switching from virtual component to real component, the runtime support system can smoothly move to the real control system. However, when the runtime support system connects to the real control system, there is some extra information that needs to be added on:

- Hardware properties. In real components, some hardware specific properties need to be added into the runtime support system. Although the interface has been

defined in the component interface specification, it is quite difficult to simulate in the virtual environment and is not necessary. In the real world control system, more information is provided to the runtime support system than in the virtual world. Especially in terms of configuration, it can only test in the real world.

- Exception handling and diagnostics. These two items need be refined in the real control system. As mentioned before, the control logic will be further developed in order to handle unexpected problems. Therefore, the runtime support system also needs to be further developed, in order to confirm the control logic in the real world.

Although there is some work, which still needs to be carried out after the real control system is ready, compared to the traditional way, it really promotes development speed.

However, after the control system is delivered, the runtime support system can easily upgrade the control system as shown in figure 6.6. Since the whole control system is component-based, the runtime support system can upgrade the system by replacing a component by an appropriate new one, which may be developed by CSDE or a third party. The 'appropriate one' means that the component interface needs to be compatible with the old one. The new component maybe adds in some new interface items, but at least it must confirm the old interface. The configuration interface of the connector that was described in chapter 5, facilitates this activity. This activity is a part of the runtime support function.

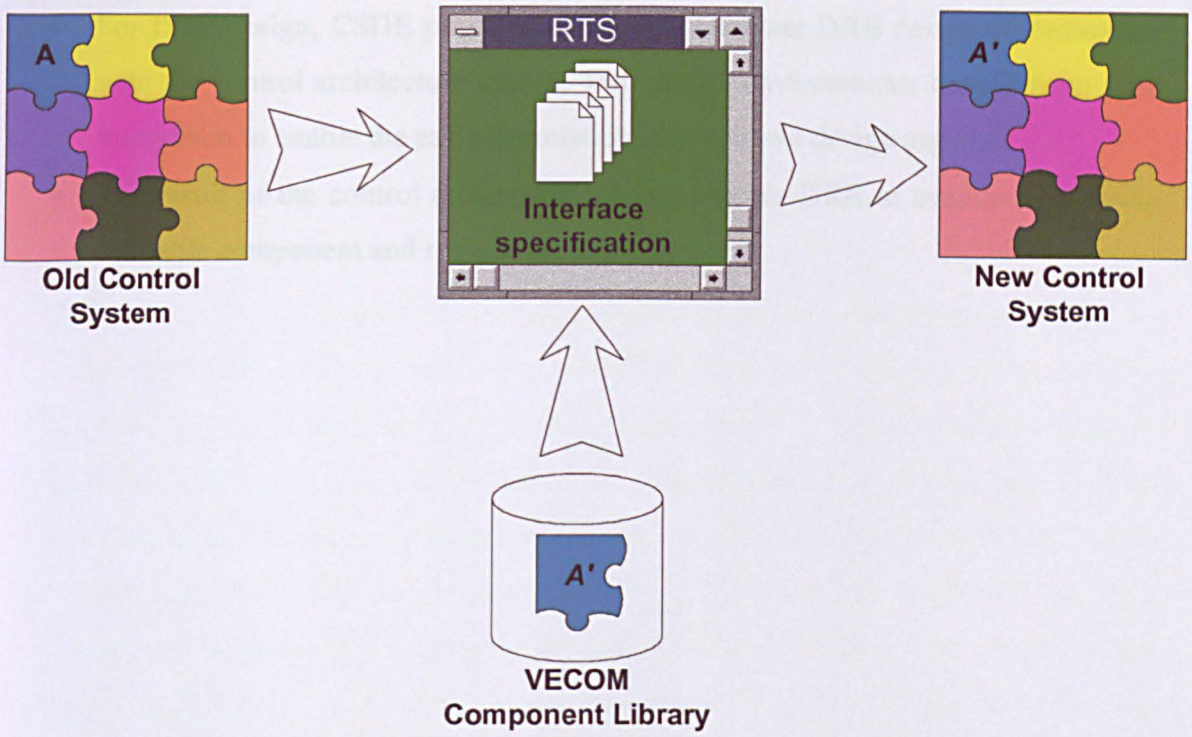

Figure 6.6 Configuring the Control System

6.5 Summary

The integration issue is addressed by referring to the VIR-ENG machine design process, which is specifically devised to provide the guidelines for efficient use of the VIR-ENG environments in supporting the machine design and implementation life cycle. Consequently, the integration of CSDE is identified as the interaction with MMDE and DRE. Integration with MMDE is achieved in two aspects:

- At the conceptual design stage, UML-based diagrams and tables facilitate the exchange of designs between the two environments. The architecture-based development approach facilitates later component development.
- At the implementation stage, the component-based development approach enables control system development to switch between the two environments, depending upon the specific development phase requirement. One of the major benefits from integration between the virtual and real environments is the use of the simulation model to verify control system design before it is actually applied to the real machine system.

The integration with DRE is achieved in two aspects:

- For DRE design, CSDE provides the tool to facilitate DRE design co-ordination with the control architecture design. Two design environments benefit from such integration to enable the early negotiation between two design aspects.
- The result of the control architecture design enables DRE to evaluate the newly available component and replace the component.

Chapter 7 Demonstrator Cell

7.1 Introduction

Realisation of the agile manufacturing machine system reflects the major objective within the ESPRIT IV project 25444 - VIR-ENG, "Integrated Design, Simulation and Control of Agile Manufacturing Modular Machinery", where engine assembly production is investigated. It has led to the construction of a demonstrator cell by Euromation in Sweden to demonstrate that the VIR-ENG concepts are practical and to evaluate the VIR-ENG design environments. Typical machine types for assembly production systems (e.g. assembly machines, modular conveyors, AGV, gantry loader, etc.) have been collected in the demonstrator cell. One of the major parts of the programme is the demonstration and evaluation of the CSDE methodologies and associated tools within the demonstrator cell.

The agile modular manufacturing machinery paradigm reflects the requirements of engineering enterprises: rapid introduction of new machine systems with proven functionality, rapid reconfiguration of existing modular machines systems (or manufacturing) facilities, simplified and proven retrofitting with minimum disruption to accommodate minor design changes in the product (or process) and quick product change-over for predefined feature (or model) variants to facilitate small quantity production volumes. Based on these arguments, three test scenarios have been designed to assess the capabilities of the VIR-ENG methodologies and tools. Furthermore, another important point of the assessment is to evaluate the incorporated multidisciplinary design, which refers to agility in the context of the design process. As the focal point of this research work, this chapter concentrates on the control relevant issues, which are mainly influenced by the CSDE methods and associated tools. The test cases are designed as follows:

- The first scenario was to create a new production cell from scratch, which goes through every step of the proposed workflow.
- The second scenario evaluates the capability to cope with product change over.

- The third scenario examines the capability to deal with introduction of product variants.

It is perhaps worth stressing here that although there are many research projects addressing similar issues, only a very few have been reported that have successfully built a system to scale in an industry application fashion. This highlights the achievement of "VIR-ENG", in the sense of successfully building a working system to scale. A significant proportion of this achievement comes from the control system design and development aspects, to which this research makes a direct contribution.

This chapter describes the three test cases. Since the first case involves every phase of the design process actions, it is divided into six sections, which describe the construction of a new production cell for a 5-cylinder head valve assembly and evaluates of results. Based on this new production cell, the other two cases focus on the additional effort needed to achieve cell variants as defined above.

7.2 Inception Phase

With reference to the design process that is described in chapter 3, the whole VIR-ENG design activity starts with the inception phase, which includes user requirements analysis, conceptual design, simulation concept, mechanical design and simulation of the mechanical aspects.

7.2.1 User Requirement Analysis

Because there was no project available that could be used as a pilot within the ordinary Volvo production facilities, the demonstrator cell built was based on the availability of facilities provided by Euromation, some facilities being old ones. The configuration and layout of the demonstrator was chosen to be representative for typical equipment that is normally found in the industrial production environment. The main parts of the cell configuration include:

- A gantry loader for material handling purposes.

- A conveyor system for conveying.
- An AGV for flexibility in far distance transport.
- An assembly machine representing the first machine in a series of machines in a complete line.
- A manual workstation.
- A central safety system.

The products chosen are cylinder heads used in some models of the VOLVO S70 and V70. The physical layout is shown in figure 7.1. The physical features of a cylinder head are:

- The cylinder head weight is approximately 20 Kg.
- The size of a 5 cylinder head is (L x W x H): 531 mm, 275 mm, and 129 mm.
- The total weight including the pallet for transport on the conveyer is approx. 25 Kg.
- The weight of the valves is approximately 50 g each.

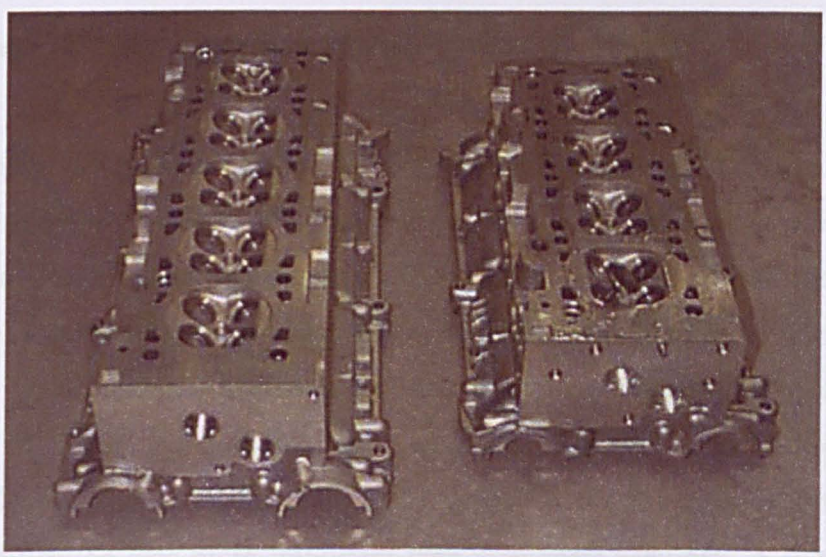

Figure 7.1 Two Types of Cylinder Head

The demonstrator should feature a precision mechanical assembly operation representative of applications in automotive engine assembly. Furthermore, the complete system would incorporate the major manufacturing processes including materials handling (part distribution, loading, unloading); inspection; parts

transportation; safety systems; etc. Figure 7.2 shows a valve partly inserted into a cylinder head.

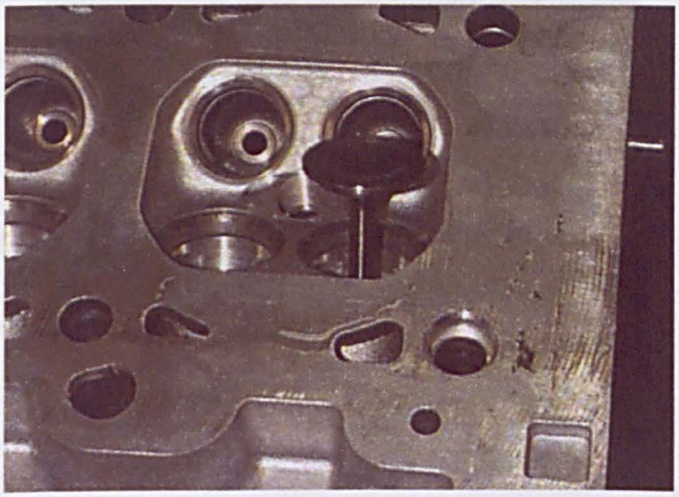

Figure 7.2 The Valve in the Cylinder Head

7.2.2 Conceptual Design and Simulation

The conceptual design and simulation concept started with the machine system layout design. Initially, the design was carried out using 2D drawings, which quickly effected some important decisions, such as the size of the machines and the position of machine. After the 2D layout has been proved, the model is further developed in a 3D-design environment. Meanwhile, the conceptual simulation takes place, illustrating the main operation in order to meet the essential manufacturing process via MMDE tools. One example is shown in figure 7.3. At this stage, it is evident that concept verification was more important than geometric accuracy. A more accurate geometry model will be available after the mechanical design.

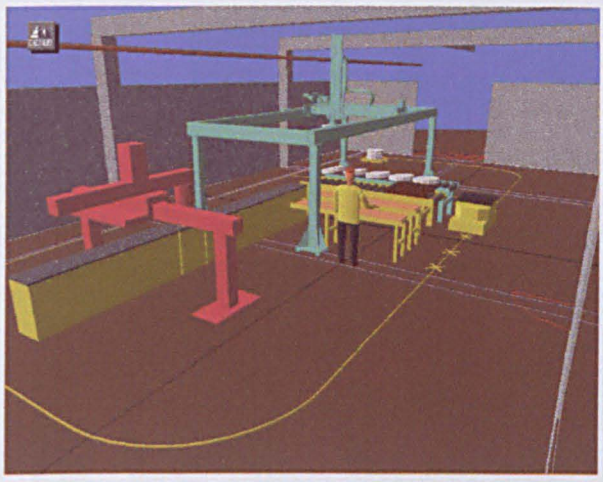

Figure 7.3 The Conceptual Model of Demonstrator

7.2.3 Mechanical Design and Simulation

The mechanical design started with the development of 3D models of the different parts within the demonstrator. The modelling work was achieved through the combination of using the simulation tools in MMDE and importing some models from external CAD software. Subsequently, accurate mechanical models were available for simulation of mechanical aspects. In some aspects, the models were simplified to improve performance. On the other hand, the mechanical design of the modular machines had to contain enough detail for them to represent the real system.

The main activities carried out in the simulation of the mechanical aspects are to describe the event logic from a mechanical perspective and/or process perspective. Through executing the models, the simulation validated sequences of the moving parts against pre-defined motion profiles. The event logic is a rough sketch that describes the operations of the assembly machine, without concerning the control architecture of the machine, although, the event logic is using SFC to describe. Figure 7.4 shows the IGRIP model combined with the programming environment and the VEVIO viewer. The verification of the event logic is based on observation of the logic flow in the programming environment and the simulation model in IGRIP. To test the event logic, the user can view and manipulate the I/O values through the VEVIO viewer.

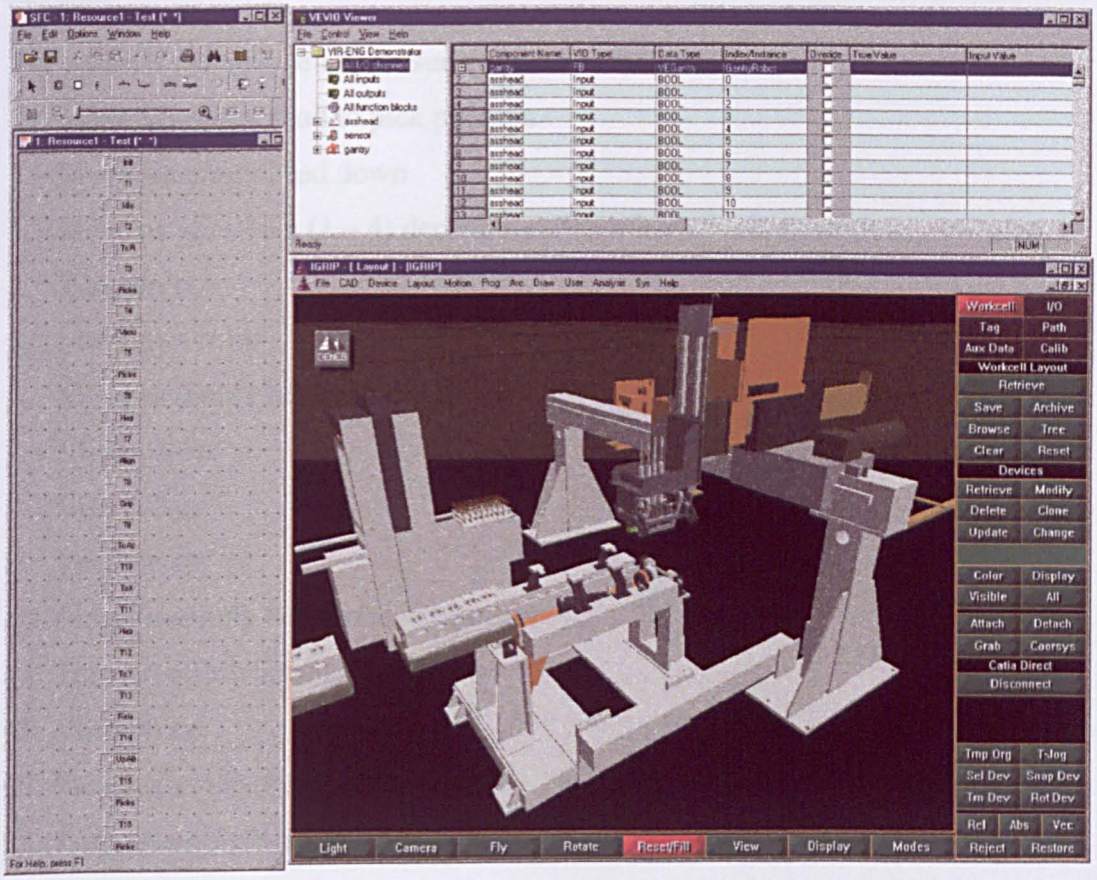

Figure 7.4 Event Logic Simulation in MMDE

7.2.4 Control System Requirement

After the inception phase, most of the control system requirements are available, which include the legacy system identification, the event logic descriptions, and mechanical specifications.

There were three different kinds of legacy systems in the demonstrator, which are the valve magazine, gantry loader, and AGV. The valve magazine was originally used for sparkplugs but was easily converted to handle valves; it is controlled by a Siemens PLC. The gantry loader manufactured by Euromation can handle and transport work-pieces and pallets, and can lift up to 500 kg. The control system in the gantry loader was a Siemens S7 PLC, namely S7 315-2DP with additional I/Os. The positioning system for the four axes was from SEW, namely EuroDrive. The AGV was built by Euromation, and controlled by a central computer. The radio LAN provided the communication mechanism to receive commands from outside.

The assembly sequence can be summarised as follows:

1. Move assembly head to pick position
2. Move assembly head down
3. Move picker down (1 – 4) depending of number of valves needed or existing
4. Vacuum on
5. Move picker up
6. Move assembly head up
7. Align valves
8. Grip valves
9. Vacuum off
10. Move assembly head to load position
11. Turn assembly head to x position
12. Vacuum on
13. Move assembly head down
14. Turn assembly head to y position
15. Release valves
16. Vacuum off
17. Move assembly head up a bit
18. Move picker down
19. Move picker up
20. Move assembly head up

The material flow in the demonstrator can be described as shown in figure 7.5. The figure below shows the names and layout of the conveyor modules, to assist the subsequent process description. The system assumes an external source exists that supplies or collects cylinder heads to / from the VIR-ENG AGV (e.g. a manual forklift).

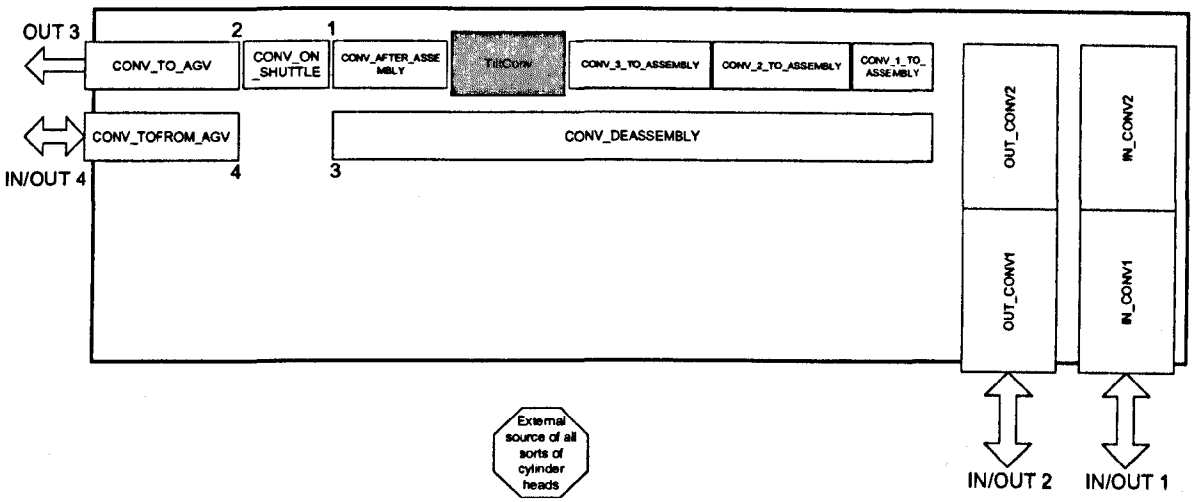

Figure 7.5 First Scenario System Layout

The system was designed to handle a 5-cylinder (5-cyl) cylinder head (CH hereafter). CHs components are fed into port IN 1 (i.e. the IN_CONV1) by a forklift driven by an operator. They are loaded to CONV_1_TO_ASSEMBLY by the gantry robot and towards assembly. After assembly, all assembled CHs are conveyed to the inspection/disassemble station through the shuttle. The operator in this station performs inspection and disassembly of the CH as necessary. By pressing different buttons, the inspection operator identifies a bad or good assembly to the relevant participants. Disassembled CHs are transported back to CONV_1_TO_ASSEMBLY for re-assembly via the gantry robot. All good assemblies are picked up by the gantry and then placed on the pallet to wait on OUT_CONV2. When the pallet is full (four CHs), the pallet will be moved to OUT_CONV1 (i.e. OUT port 2) and is then removed by the manual forklift.

In the ordinary operation condition, an AGV is used to convey assembled 5-cyl CHs, which were assembled in another work-cell, to the VIR-ENG cell for inspection. The externally assembled CH is transported by AGV to port IN/OUT 4 of the conveyor system and then the assemblies are unloaded onto CONV_TOFROM_AGV. Since the externally assembled CHs also belong the same product type, the same inspection process (as described above) can apply o these external CHs.

Consequently, the kinematics properties that can be found out are that the assembly machine axes are loosely coupled, which implies that this can be achieved by several individual controllers. The moving limit is also available from the simulation.

7.3 Control Architecture Design

Based on the description of the CRC method and associated tools described in chapter 4, the following description uses the demonstrator as an example to illustrate the design approach and how it provides supporting tools for the design. In order to simplify the content, the description concentrates on the final results rather than covering the full temporary processes.

7.3.1 Task Analysis

The goals of task analysis are:

- Define the abstract manufacturing operation that needs to be provided by the control system.
- Translate the control system requirement context into a formal description.
- Solidify these requirements by turning them into a task decomposition diagram.
- Break down the high-level requirements into a feasible basis for the control system.

According to the previous description, the main machines have been determined and the necessary operations have been chosen. Based on these facts, the task identification and decomposition can be referred to the involved machines.

At the top, the overall task is the production process, which is finally decided by the machine end user. It is difficult for the machine builder to determine the operation arrangement. To design the machine system, a standard operation arrangement provided by customers was adopted to design and test the system design. After the entire system has been delivered, the end-user may rearrange the operation process depending on the manufacturing requirements.

Below the overall task, there is a set of tasks: AGV operation, 'Pick' n 'Place' operation, conveying operation, and assembling operation, which describe how the global task can be achieved. The AGV operation involves one AGV, which feeds a single cylinder head into the system or removes a single cylinder head from the system. It is the link between the system and the outside environment. But it was also supposed to deal with some non-standard cases. For example, the flawed cylinder head that is assembled by another system is fed into the system to be disassembled. The batches of cylinder heads that are fed in and removed are handled by the manual forklift, which is excluded from the system. The conveying operation including the conveying system sections and cooperating with the gantry loader establishes the resource flow from receiving the batch of unassembled cylinder heads to the batch of qualified assembled cylinder heads. The manual assembled cylinder head inspecting and disassembling is also considered as part of this. The gantry loader facilitates 'Pick' n 'Place' operations. All of these are considered as a concurrency operation. That means there may be several cylinder heads in the conveying system and operations may occur at the same time. Assembling cylinder heads is the key part of the system, which includes an assembling machine and a magazine. It inserts valves into the cylinder head; the number of valves depends on the cylinder head type. These operations are sequence actions; during one operation cycle, only one cylinder head should be dealt with.

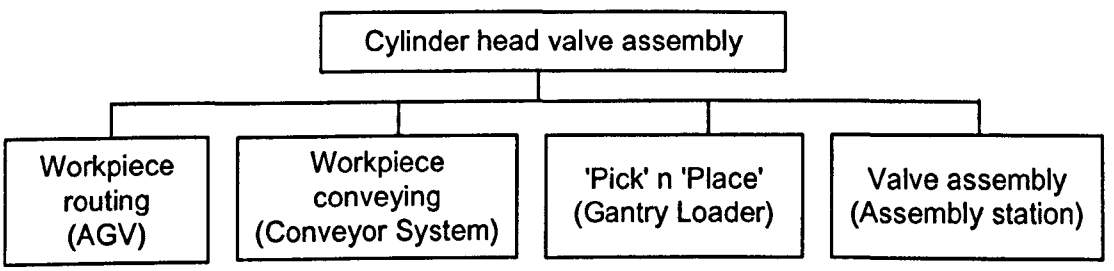

Figure 7.6 Top Task Decomposition

In the next level, the AGV is available in the first place; this is considered to be a legacy machine based on the fact that the interface is not compatible with the VECOM model. Thus AGV operation has not been further decomposed.

The gantry loader is considered as another legacy machine, whose operations are transferring cylinder heads between the conveyor sections. Based on the same consideration, the operation was treated as a whole.

The further task decomposition of the conveying operation is shown in figure 7.7. Although inspecting and disassembling assembled cylinder heads is achieved manually, the result of the actions needs to be fed back to the system and affects the system operation. These two operations have been taken into account. Shuttle operation is the combination of shift operation and conveying operation to transfer cylinder heads to a different destination. It has been separately considered and integrates a conveyor and a position switch. The remaining conveying operations are categorised into two types based on the capability of the entity number handled. Most conveyors only deal with a single cylinder head, which buffers single cylinder heads (one engine block exits one conveyor section). In contrast, the belt conveyor that is CONV_DEASSEMBLY can handle more than one cylinder head, which is buffering multiple cylinder heads (it could move more than one cylinder head in one conveyor section). At the next level of decomposition, the identified operations are decomposed into the same child tasks. These tasks are small enough to be feasible. Therefore, that is the end of the conveying operation decomposition.

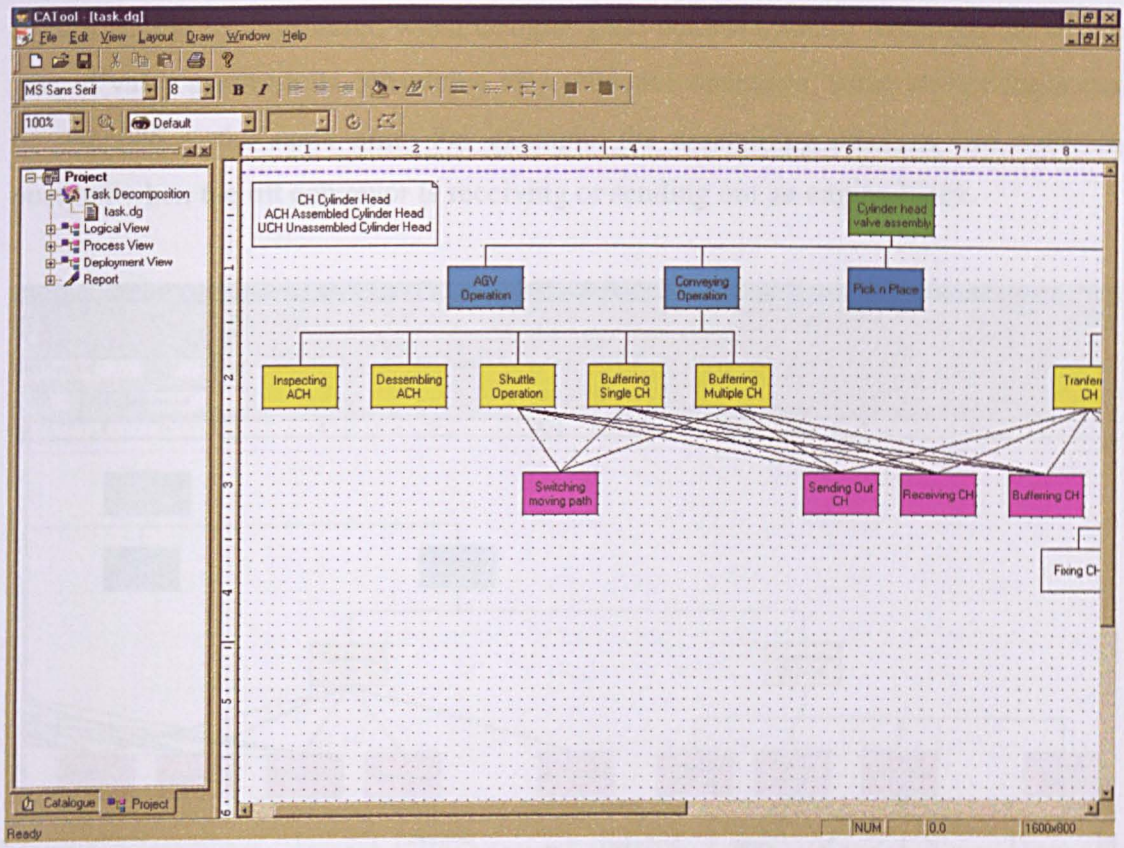

Figure 7.7 Conveying Operation Decomposition

Valve assembly decomposition is shown in figure 7.8. It is divided into two child tasks, which are transferring a cylinder head and the assembly operation. Transferring a cylinder head involves the tilt conveyor, which ensures the position of the cylinder head for the assembling operation. Assembling operations are facilitated by the assembling machine, which enables assembling functions. These two tasks are further decomposed into essential tasks as the figure depicts. There are three atomic blocks, which are receiving cylinder head, sending out cylinder head, and buffering cylinder head the same as the conveying operation. It should be noted that there are several tasks related to the same mechanical element. For example, the fixing engine block and unfixing engine block are achieved by using the index. Magazine operation is considered as another legacy machine that is magazine, which responds to change pallet to provide enough valves for the assembling machine. Magazine operation involves some manual operations. If there is no full pallet, it will send a signal to notify the operator so that he feeds a new full pallet and removes the empty one. Several detecting error functions are also embedded in the magazine controller. These

signals should be considered when designing the control system. Although the whole task of valve assembly is considered as a sequence operation, some atomic tasks can be operated at the same time. For example, the assembling machine can move to position when the tilt conveyor is receiving or sending out an engine block.

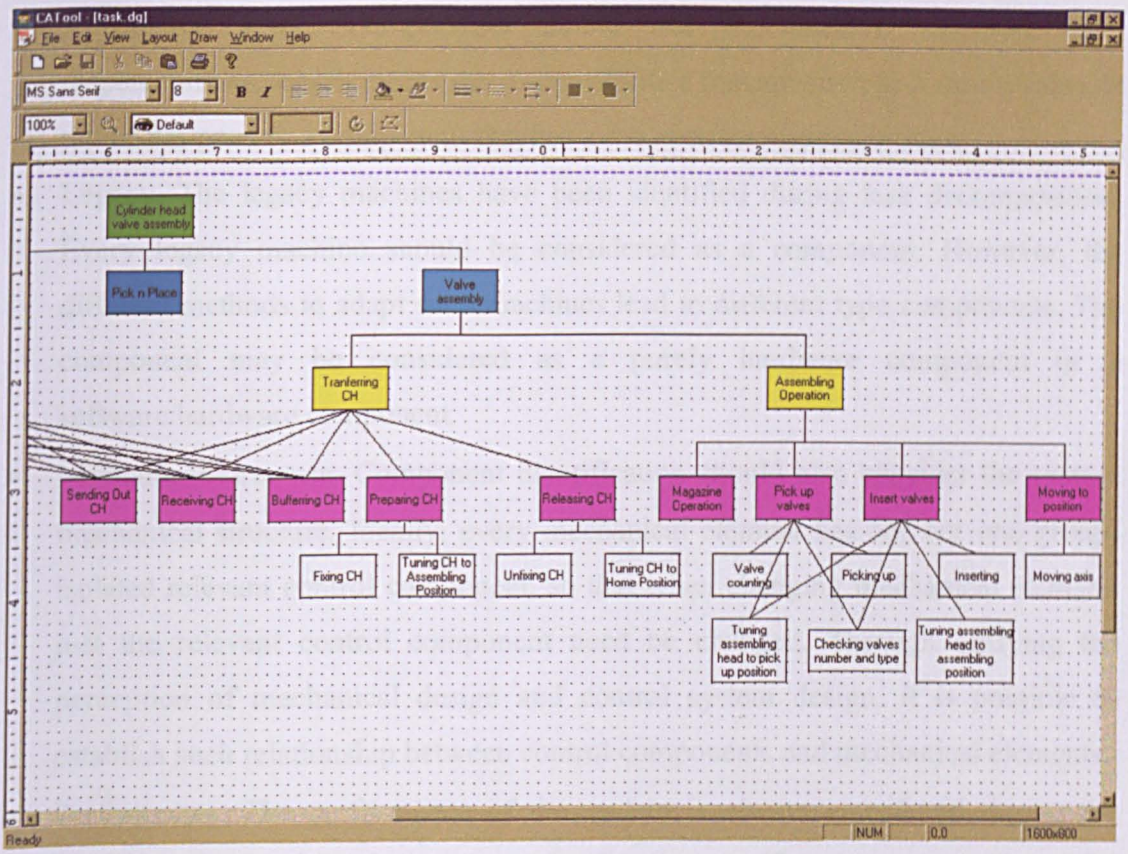

Figure 7.8 Valve Assembly Decomposition

7.3.2 Component Identification

Through the task decomposition, the main responsibilities of the control system, which need to be achieved, have been identified. These responsibilities should be gathered into groups and assigned to the component. As described in chapter 4, the deployment view is used to identify the components. Figure 7.9 depicts the whole system deployment view. Several criteria have been followed during the component identification process. They are:

- Functional coherence. Responsibilities that are assigned to components should exhibit low coupling and high cohesion. This is a standard strategy for component identification to allow components to be readily upgraded and reused. Some existing technologies can support such analysis. For example, the use cases can be used to investigate the cohesion and coupling.
- Abstraction level. Components should not be assigned different level responsibilities, which means the responsibilities that are close to hardware should not be assigned to a component that has more abstract responsibilities.
- Legacy. The legacy machines have been identified during task decomposition. Every legacy machine should be considered as a component. However, the different methods to adapt such machines lead to different type components. The component may be considered as a purely hardware component or a software/hardware component.
- Mechanical structure. Component identification should also consider the system mechanical structure. Ideally, every mechanical element has one (or one group) isolated relevant control component, so that when changing mechanical element, just the relevant control component need be changed. Through unifying the perception of mechanical design and control system design, it is possible to establish such relationship between control components and mechanical elements.
- Performance. The performance of the system is always considered during the whole component develop process. In some cases, if the system performance affects the system quality, then the division needs to be based on minimising the data exchange between components.

The architecture style confirms the VIR-ENG four-layer model, as shown in the deployment view. On the other hand, the layer architecture also reflects the physical layout of the system.

The machine Systems layer, shown as the top layer of figure, is at the highest layer of the architecture. It is supposed to provide some standard operations that are common across the machine system. Normally, there is only one control logical component in this layer. The run-time support components are also included in this layer, which are

not shown in this figure. The responsibilities of the component in this layer can be summarised as in table 7.1.

Component	Responsibility
Cell co-ordinator	• Provides interfaces for runtime support to access control functions and system information. • Initialises the system component with a pre-defined configuration. • Route requests/messages to components in the lower layer to co-ordinate the modular machines. • Implementing application specific task • Manage system-wide information. • System error handling • Responding to mode changes

Table 7.1 Machine System Component Responsibilities

The modular machine layer, shown as the second layer, represents the identified machine. The components in this layer consist of three types of machines, which are AGV, conveying system, and assembling station. The responsibilities of these three components are defined in table 7.2.

Component	Responsibility
Conveying system	This control component coordinates the activities of different composite components to transport workpieces within the demonstrator. Its responsibilities include: • Provision of interfaces for higher-level control components to access its information and services. • Provision of interfaces for runtime support to access the control functions and machine information. • Initialises machine components with a pre-defined configuration. • Holds various pre-defined transportation routes. • Acts as a bridge for communication with the gantry loader • Routes requests/messages to components in lower layer. • Manages the machine-wide information. • Machine error handling
Assembly station	This control component coordinates the activities of different composite components to assemble valves into the cylinder head. Its responsibilities include: • Provision of interfaces for higher-level control components to access its information and services. • Provision of interfaces for runtime support to access the control functions and machine information.

	• Initialises the machine components with a pre-defined configuration. • Routes requests/messages to components in lower layer. • Implementing assembly specific task • Manages the machine-wide information. • Machine error handling • Responding to mode changes
AGV	This control component provides a wrapper to enshroud the original interfaces of the AGV without modification. The wrapper presents the AGV as a VIR-ENG component to the cell co-ordinator. The services of the AGV could be found on its manufacturer's data sheet and manuals. It has the following responsibilities for the demonstrator: • Transferring messages between the AGV and cell co-ordinator • Responding to manual commands

Table 7.2 Modular Machine Component Responsibilities

The composite component layer, shown as the third layer, comprises of the device components. A component that is located in this layer provides the atomic control action. The atomic control action can be obtained by means of grouping the tasks and identifying similar data or computation patterns. By providing a meaningful service rather than simple signal or data, composite components can be easily composed from the higher-level components. On the other hand, this layer provides flexibility in the face of changing hardware components. In detail, the components responsibilities are defined in table 7.3.

Component	Responsibility
VEConvComp1	• Provides generic single direction conveying operation
VEConvComp2	• Provides generic bi-direction conveying operation
Conveyor1	• Provides single direction conveyor information
Conveyor2	• Provides bi-direction conveyor information
Shuttle	• Provides pallet status • Coordinates conveyor and cylinder operation
TiltConveyor	• Coordinates conveyor and tilt operation • Provides internal status • Responds to mode changes • Supports manual operation • Error detection and recovery
Assemblyhead	• Coordinates cylinder head operation • Provides internal status • Responds to mode changes

	• Supports manual operation • Error detection and recovery
Pneumatic	• Provides main air and vacuum compressor operation • Provides internal status • Error detection
Cabinet	• Provides cabinet status • Provides assembly station power modular operation • Error detection
Gantry co-ordinator	• Coordinates three axis movement • Provides internal status

Table 7.3 Composite Component Responsibilities

The bottom layer is the device component layer. In general, this layer consists of device components, which could be software/hardware components such as SDS smart sensors or pure hardware components such as pneumatic actuators. The pure hardware identification provides sufficient information to construct the virtual environment and develop or select hardware elements. Table 7.4 defines the responsibilities of device components.

Component	**Responsibility**
Actuator	• Provides actuation.
Sensor	• Provides data for monitoring and control.

Table 7.4 Device Component Responsibilities

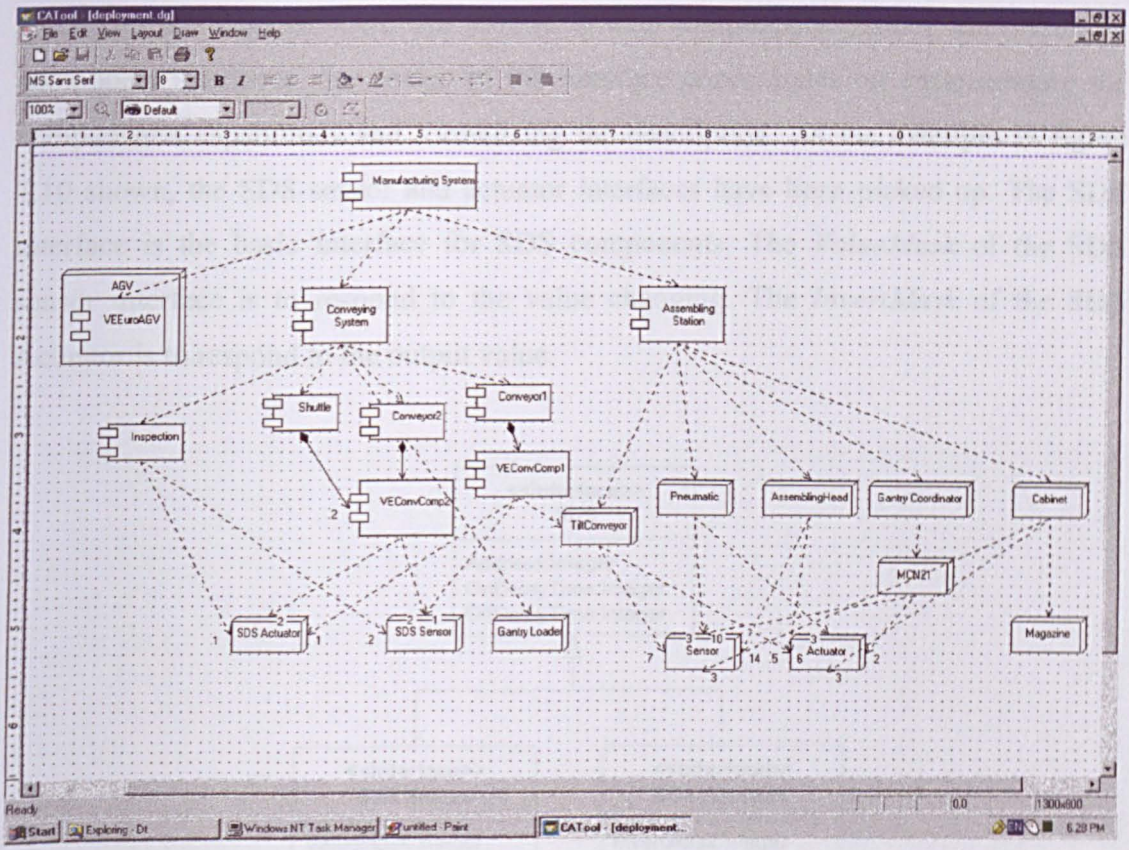

Figure 7.9 Deployment View

7.3.3 Interface Specification

The interface specification relates to two aspects. While the logical view describes the static details of the component interface, the process view describes component dynamic behaviour. Since these two views effect each other, the logical view and the process view should be considered simultaneously in terms of determining the interface of components.

A bottom-up design is adopted during the design process. The bottom-up design process means that design from device components to machine system component is recommended. The reason is that, compared to high-level components, the tasks or responsibilities of low-level components are simpler and there is less collaboration between components. In addition, interfaces of low-level components are general and determined by capabilities.

As figure 7.9 depicts, there are several device components. Rather than defining component interfaces, the design of the interface concentrates on encapsulating the component's capabilities and formalising the description. As the example in figure 7.10 shown, the SDS sensor and actuator interfaces have been picked up. The SDS interface is the basic interface for SDS components. The *ValueMask* of the SDS sensor interface is to respond to the value changing. The *InvertMask* of the SDS Actuator is to respond to the output value.

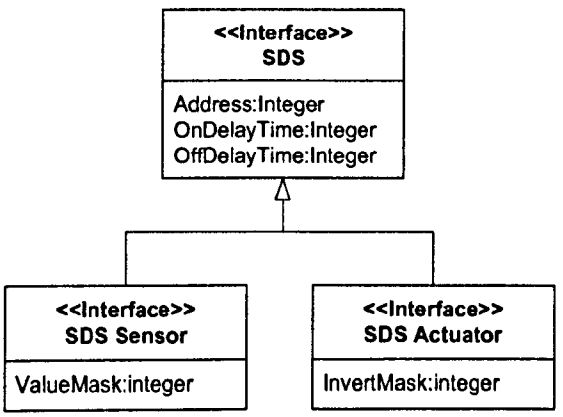

Figure 7.10 SDS components

At the high-level, the components are software intensive components. In other words, the high-level components focus on the software interface design. During the interface design process, the common interface is intended to be defined early, so that the design and implementation effort can be reduced. As a part of understanding the specification of the functions for the above components, the high level component should play several basic roles, which respond to change control mode, inform errors, initialise service and shutdown service. This leads to describing a common interface, that is the VECOM interface, to group these basic roles. In order to more simply describe the interface specification and reuse the interface specification, the other components' interfaces will aggregate this interface as basic items of their interfaces. In figure 7.11, VEConvComp2 provides service for the bi-direction conveyor element and VEConvComp1 provides service for the one direction conveyor element. It is evident that the bi-direction conveyor element has one additional sensor than the one direction conveyor. So that the VEConvComp2 that inherits from the VEConvComp1

adds an additional interface item to define its interface. In practice, discovering the same operation and constraints for VEConvComp1 and VEConvComp2 not only involves investigating the mechanical function, but also assists the final implementation of the component. For example, if the VEConvComp1 component already exists, the interface design may lead to a different result for the VEConvComp2 interface. Precise and clearly understandable interface specifications will greatly assist this task.

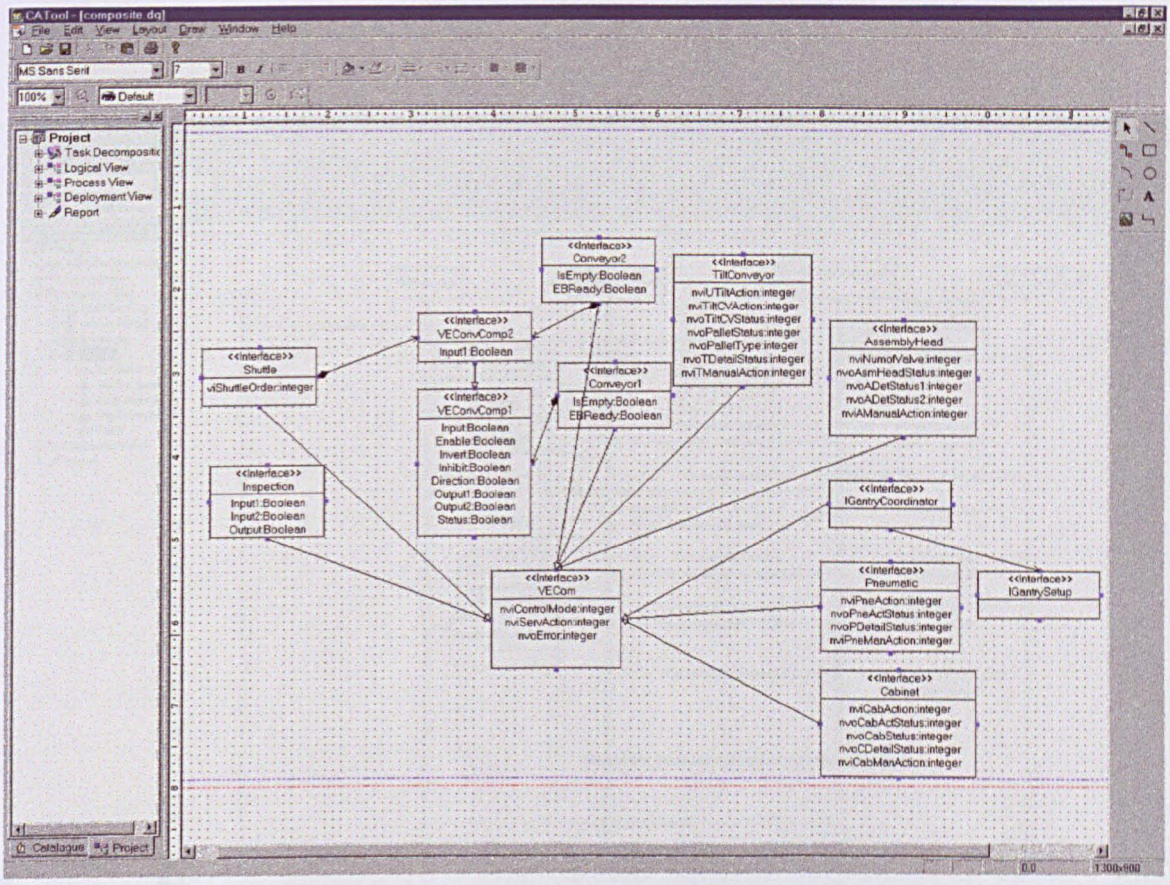

Figure 7.11 Logical View Example

The process view is another perspective on the component architecture, which focuses on the dynamic behaviour. The process view needs some relative dynamic scenarios, which are obtained from the event logic to assist the design. In addition, the result of the process view helps to refine the component interface, which is described in the logical View. An example of the process view is shown in figure 7.12. It consists of four components, which are Tilt Conveyor, AssemblyHead, Assembling Station, and Gantry Co-ordinator. Some of the component interfaces have been shown in figure

7.12. During the process to generate the logical view, some component interface operations that relate to collaboration with the other components are not specified. By generating the process view, those operations should be clearly defined. For example, in the AssemblyHead interface, there is only a name of assembly head action. Through analysing the process view, the nviAsmHeadAction can be defined as 0 – No action, 1 – Pick up valves from magazine with pickers in "right" order e.g. 1,2,3,4, 2 – Align and brace valves, 3 – Turn unit to assembly position and grab the valves, 4 – Insert valves, 5 – Check insertion of valves, 6 – Pick up valves from magazine with picker in opposite order e.g 4,3,2,1.

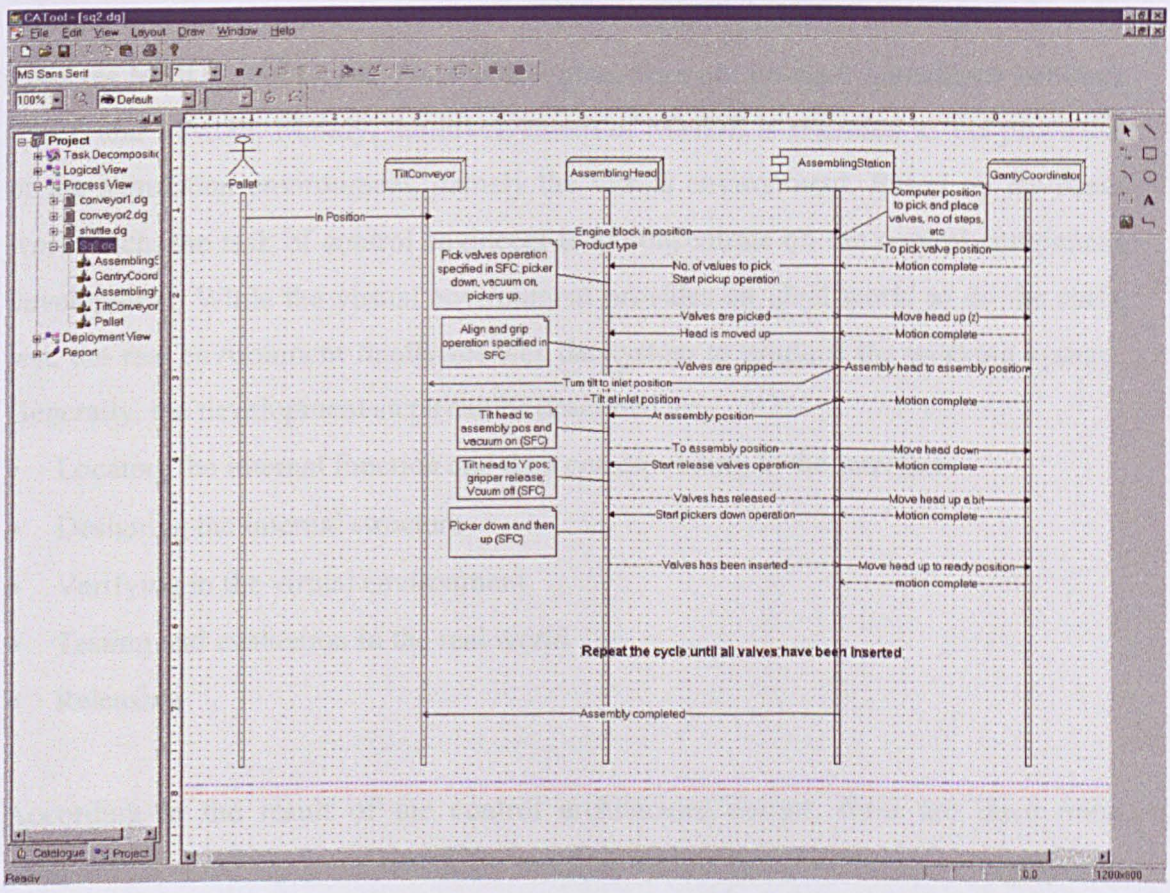

Figure 7.12 Process View Example

The activities for generating logical and process views cannot be completed during one design cycle in the same as the deployment view. In practice, some decisions may be made after some detailed design stages. However, the result of every design cycle always contains valuable information for implementers.

7.4 Control System Implementation

After the completion of the control architecture design, several participants use the design results. They are Modular Machine Design Environment (MMDE), Distributed Runtime support Design Environment (DRE), Component Design Environment (CDE), and Control Logic Programming Environment (CLPE), respectively. CDE copes with the very low-level component development, which is concerned with the constraints of the environment that is the low computing power and the low physical resource. DRE uses the results as the constraints to provide the information for the operator and the high-level controller. The control system design and implementation involves MMDE and CLPE. For the designer, there are no clear boundaries between CLPE and MMDE. Mainly, the involvement of MMDE is regarded as the provision of the simulation environment, namely the virtual environment. Based on the basic workbench, the task of control engineers is to concentrate on the control component development. While the virtual environment provides an ideal world to do the early test, the real environment finally realises the system to produce the working system. Generally, the development steps can be described as:

- Locating the internal function and data corresponding to the interface
- Designing the internal structure
- Verifying in the virtual environment
- Testing and evaluation in the real world
- Releasing

According to the result of the control architecture design, there are three main components at the top level, which are machine system component, conveying system component and assembly system component. Initially, they were composed in the single project to establish the clear boundaries and relationships, as shown in figure 7.13. During the process, the skeleton codes and variables inside the structure that describes the component interfaces between the components were also generated. Some of these codes and variables involving the use of other components were generated with VCon assistance. Some simple components, such as VEComConv1

and VEComCon2, were implemented in FB, which were constrained by system performance and resource cost.

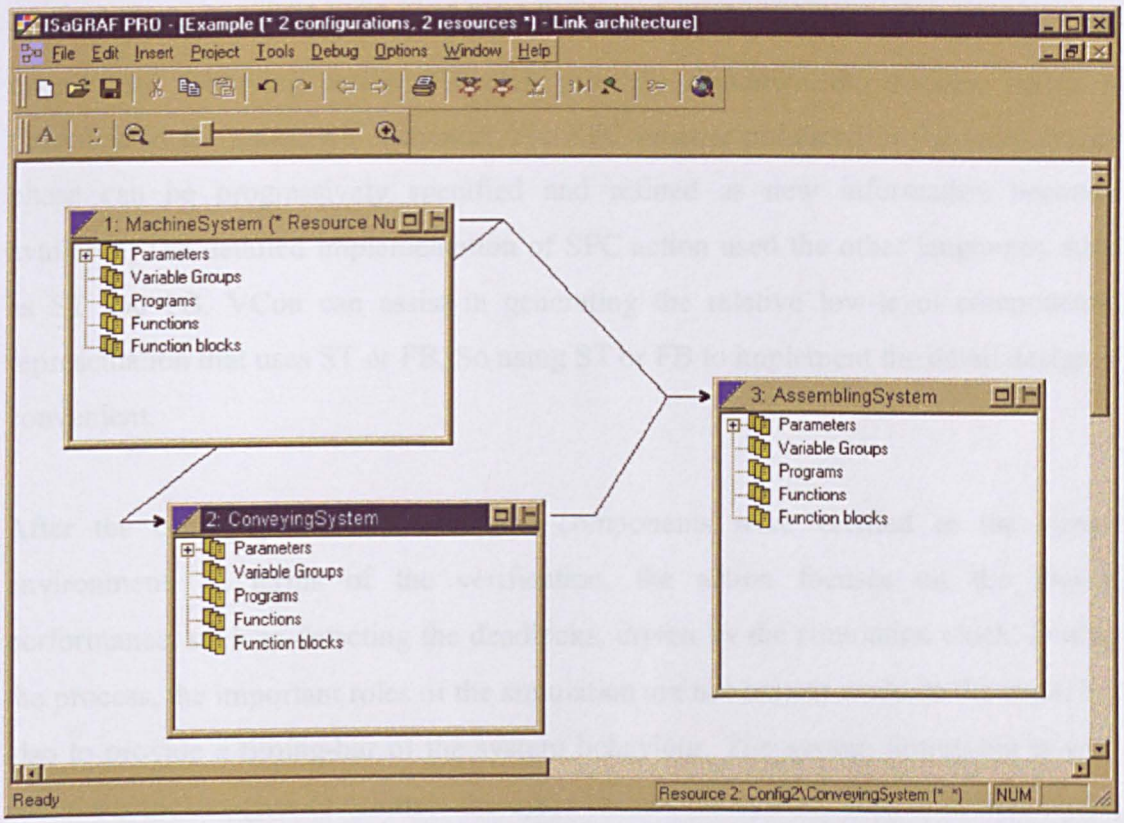

Figure 7.13 The Resource Unit Structure

After that, the resource units were separated into several independent ISaGRAF projects, so that the control components can be concurrently developed. Inside each individual project, a template has been provided, which consists of the pre-defined program structure. The main feature focuses on the main SFC, which contains three pre-defined elements S1, T1 and S3, as shown in figure 7.14. Basically, they enable the basic component operations. It also specifies some properties of project, such as the kernel mode. (There are three kernel modes relating to different resource costs.)

Later, the dynamic behaviours of the components were synthesised into SFC programs. As the example of the conveying system component shows, S2 is the action to handle the component initialising action. After the initialising action, S20 and T21 combine to form the waiting stage, to wait for the mode changing from manual to auto. In automatic operation, S5 to S15 are in change of the normal action. They

compose the in assembly conveyor operation, the out assembly conveyor operation, the shuttle operation, the disassembly conveyor operation, and the error detection operation. These actions concurrently run in normal operation to realise the conveying system operations. After the control mode changes from auto to manual or an error occurs, the following action S18 will properly shutdown the system, which is described in the shutdown sequence. The SFC scheme produced in the early design phase can be progressively specified and refined as new information becomes available. The detailed implementation of SFC action used the other languages, such as ST and FB. VCon can assist in generating the relative low-level components' representation that uses ST or FB. So using ST or FB to implement the detail design is convenient.

After the coding phase, the complete components were verified in the virtual environment. In terms of the verification, the action focuses on the logical performance, such as detecting the deadlocks, driven by the simulation clock. During the process, the important roles of the simulation are not only to evaluate the code, but also to provide a timing-bar of the system behaviour. The system timing-bar is very useful for the designer to estimate the cycle time of the system. This information also helps to preliminarily design the error handle.

The next step is to connect the program into a real component for testing. VCon was used to roughly verify the low-level components by testing worst-case execution time. Because the processor load was different from the final one, this result became only a reference. After configuring the connector, the program can communicate with the low-level components; in other words, the program can test in the hardware in a loop condition. The performance was different from the simulation in the virtual world, because of the time factor. It was recognised that it is necessary for the designer to improve the performance in the real condition. In some circumstance, some actions of the program were re-designed, so that the performance could meet the requirements. Moreover, the cycle time needed to be re-calculated in the new circumstances. In terms of exception handling, several exceptions were discovered in the real world, which were not considered during the simulation. For example, we found out the lost signal in the real world and the operation time-out handling has been added to the logic. On the other hand, in the virtual world, it will never happen.

The final action to implement the component was to use CBuilder to implement the interface of the component. By using the C++ compiler, the component was completed. Before releasing the component, it is necessary to test the performance of the component. After the final component verification, it was released and ready to be used.

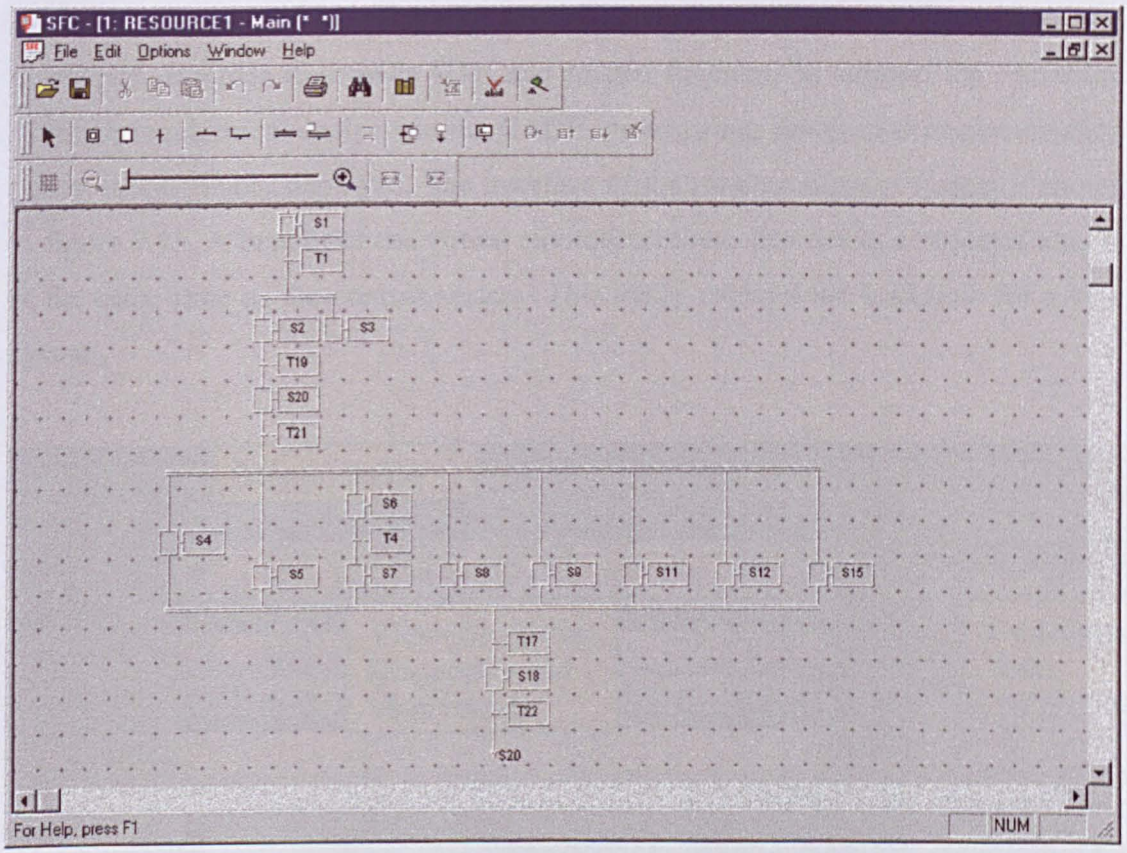

Figure 7.14 Conveying System SFC Example

Besides the control logic component mentioned above, there were some low level components. Two field-bus systems have been chosen, which are the SDS-based system and the LON-based system. Although SDS components have the computing capability, this capability does not export for user to exploit. Therefore, SDS components have been put in the lowest level in the four-layer model. In the LON system, LON nodes export the computing capability for users to develop. Therefore, LON nodes are sited at two levels, which are the composite and device level. However, these two systems' component models do not confirm the VECOM

interface. Two pieces of middleware, one of which was developed and the other was obtained from market, converted the propriety component model to a VECOM model.

7.5 Runtime Support System

When the control system was being developed, the runtime support system was also developed, in order to provide HMI and related functions to support the operation. Through requirement analysis in the CADE, prototyping, design and implementation phase, it was finally deployed. The interface of the runtime support system is shown in figure 7.15. A benefit of the virtual environment was that it was completed almost at the same time as the control system. This really reduced the lead-time for a new system.

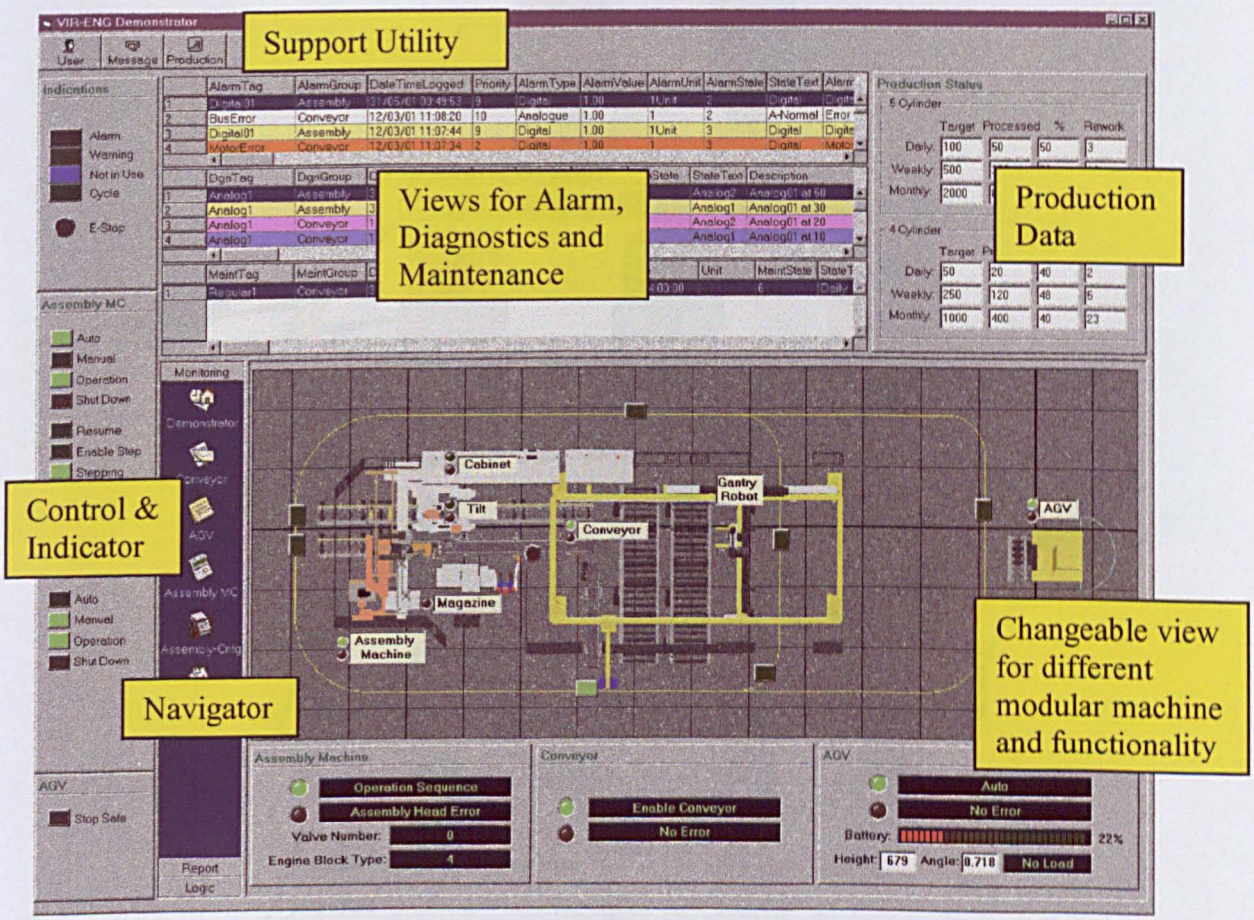

Figure 7.15 The Interface of Runtime Support System

7.6 Installation

Following the completion of the main machine elements in the demonstrator cell, they were installed in a distributed environment. The deploying structure of the final control system is depicted in figure 7.16. The runtime support and the system co-ordinator were installed at the top-level and connected via the Ethernet. Interactions between machines are governed by the system co-ordinator, supported by the runtime support. At the modular machine level, AGV mediator refers to the VEEuroAGV component, which communicates with AGV via a wireless network. The conveying system control component consisted of the SDS-based components relating to the conveying system. Similarly, the assembly system control component was composed of the LON-based components relating to the assembly station.

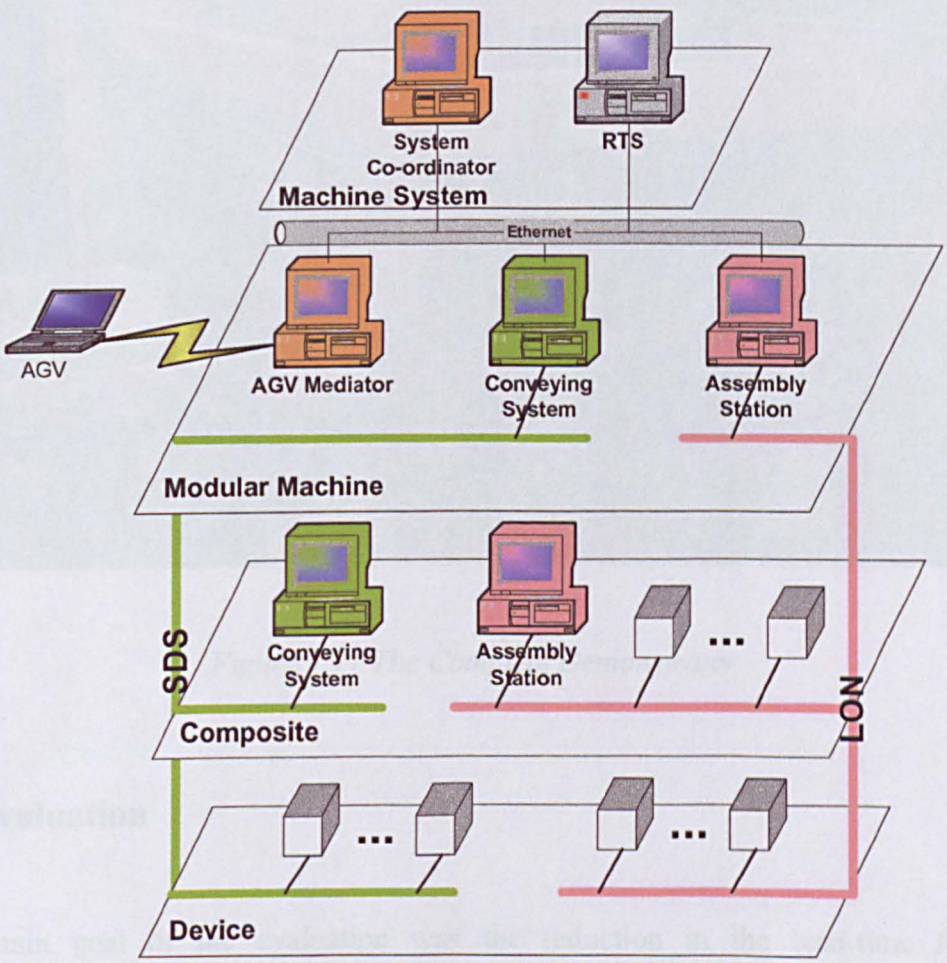

Figure 7.16 The Structure of the Distributed Environment

It should be noted that several components were actually deployed on the same computer, such as the AGV mediator and the system co-ordinator. To clearly identify this, some computer symbols in figure 7.16 are given the same colour where these are actually one physical computer. Additionally, every part of the system has been tested in order to avoid compromising the system performance. The complete demonstrator is shown in figure 7.17.

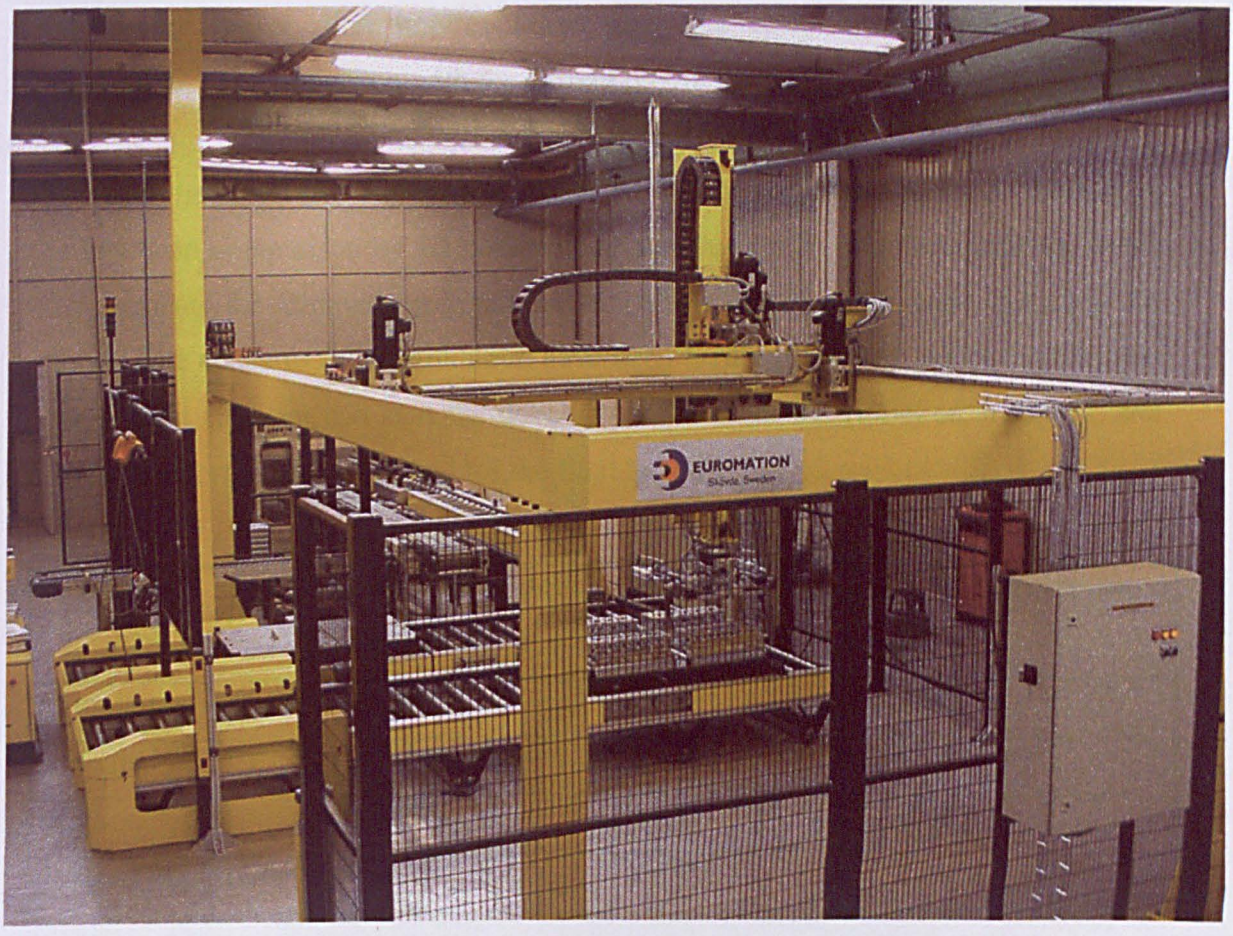

Figure 7.17 The Complete Demonstrator

7.7 Evaluation

The main goal of the evaluation was the reduction in the lead-time for the construction of a new machine system. Because of the lack of quantitative information about the relevant traditional development process time and cost, the evaluation quantifies the percentage of reuse within the development process rather

than making a direct comparison. This is considered to be indirect criteria that reflect the efficiency. In the context of the VIR-ENG, the reuse has two aspects. The first one is concerned with the use of generic components. The essential concept of component-based development is the reuse of available components, rather than developing the whole system from scratch. Here, generic components are regarded as the available components. The second aspect is concerned of the preservation during the control logic migration process from 'virtual' to 'real' environment. Higher percentage preservation implies lower time and cost. Another subgoal of the evaluation was comparison with the current production.

Because the IEC 1131-3 programs are stored in access database form, the evaluation is based on comparison of the source code size during the migration process. Table 7.5 presents the comparison result for the whole demonstrator code. As table 7.5 shows, the codes to drive simulation have been highly preserved, which implies that the control engineer can start the development work in the virtual environment, and that this effort can be preserved for the final target.

	Conveyor System (bytes)	Assembly Station (bytes)	System Coordinator (bytes)
Simulation	65,209	78,445	39,202
Control	70,102	84,626	42,155
Add-on	4,893	6,181	2,953
Reused Percentage	93.0%	92.7%	93.0%

Table 7.5 The Percentage of Reused Code

Table 7.6 displays the percentage of generic components, which are used in the whole project, and can be used in the future development.

	Conveyor System (bytes)	Assembly Station (bytes)	System Coordinator (bytes)
Generic Component	49,104	62,079	1,820
Whole project	70,102	84,626	42,155
Percent	70.0%	73.4%	4.3%

Table 7.6 The Percentage of Generic Components

Because System Coordinator is highly application dependent, it is very difficult to find generic components. However, in considering the whole system, the percentage of the system coordinator is only a small part of the system.

The operation time is also measured in the evaluation process. Actually, the operation time is 90 seconds, which is nearly twice that of the operation time in the current operation. The main reason for this was that the assembly machine is quite an old one and the speed and accuracy cannot meet the design target. In addition the control system of the assembly machine uses the LON field bus system to achieve the control. The LON field bus does provide more flexibility for the control system implementation, but the selected network rate has a low transmit speed. Network traffic problems occasionally occur; this affects the speed of the whole operation. Overall, the basic elements cause the poor performance result, which could be solved by a reasonable upgrade.

7.8 Product Changing Over

Product changeover or new product introduction is a typical manufacturing requirement. This test is designed to cope with product changes, namely changing from 5-cylinder head to 4-cylinder head with the aid of the VIR-ENG approach and toolset. In this case, the whole process remains the same. In the detail, there is no new physical element to be introduced. By re-examining the previous control architecture design, it was found out that no new component needed to be introduced. The only change required was to the number of assembly cycles, achieved by configuring the

engine block type parameter of the assembly station. Through the verification in the virtual environment, the objective can be achieved by means of configuring the property carried out via the runtime support system, without any control logic component re-design. This test case illustrated how the VIR-ENG approach and toolset readily support changes to the design of an existing production facility – a key feature of an agile manufacturing facility. The whole test case is simple, however, this case really demonstrates a new product can be easily incorporated primarily by means of re-configuring the relevant control components rather than re-designing and re-implementing a complete new system.

7.9 Introduction of Product Variants

The third test scenario considered the introduction of product variants to the demonstrator to illustrate how the system can be readily changed/modified to cope with mixed product production, facilitated by the VIR-ENG approach and toolset. The demonstrator was required to assemble 4-cylinder and 5-cylinder CHs in random sequence. The new system layout is shown in figure 7.18. Based on the previous development, a flex-link conveyor (CONV_TO_GANTRY) was added to the demonstrator to enhance the flexibility of the materials handling capability of the cell. The following modifications for the new manufacturing task can be listed as:

- A "new" conveyor (CONV_TO_GANTRY) was added with the required sensing and actuation components, which included three proximity sensors and a pneumatic stopper.
- New routing logic for the AGV.
- Additional sensing components and control logic for the assembly station to distinguish the required assembly operations and route for each assembly type.
- Additional components and application logic for the run-time support system.

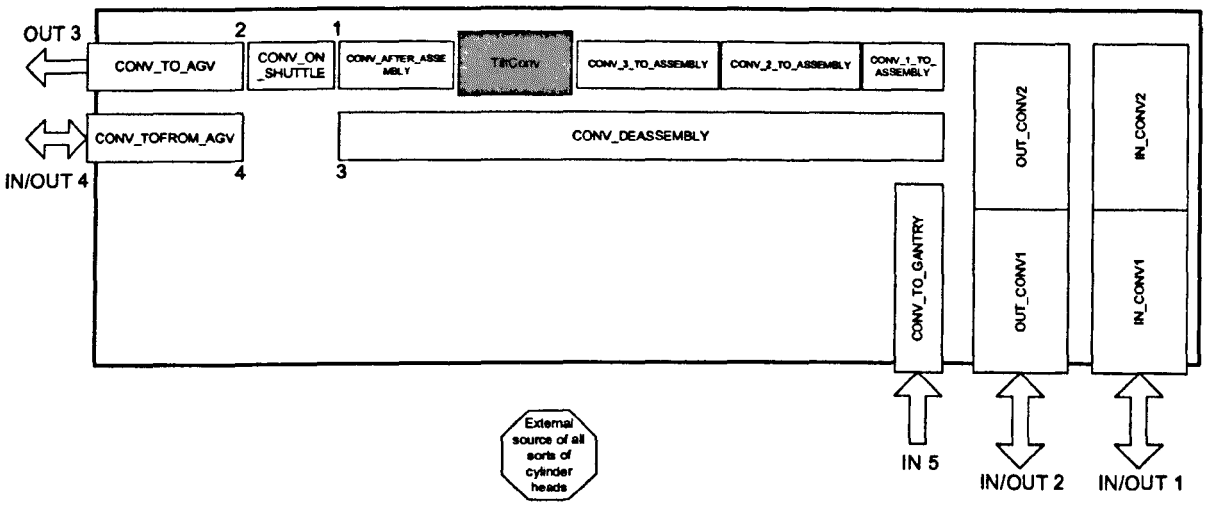

Figure 7.18 Third Scenario System Layout

The requirement for the batch production of 5-cylinder head remained unchanged. 4-cyl CHs enter and leave the demonstrator only via AGV transportation. The inspection of the 4-cyl CHs was located outside of the demonstrator. The AGV transports a 4-cyl CH to CONV_TO_GANTRY, and then unloads it onto the conveyor, when space is available. At the end of CONV_TO_GANTRY, the 4-cyl CH is picked and placed onto CONV_1_TO_ASSEMBLY by the gantry loader. There is no difference between 5-cyl and 4-cyl types for the first three modular conveyor sections, supplying parts to the assembly station. The assembly station identifies the product type through an additional sensor added to the tilt table, before the corresponding assembly operation is performed. The cell co-ordinator is responsible for signalling the cylinder head to leave the cell through port OUT 3 (i.e. from CONV_TO_AGV to the AGV). The cell co-ordinator then coordinates the AGV to pick up the assembled 4-cyl CH waiting at CONV_TO_AGV. Conversely, the production of the 5-cyl CH adopts the original routing and process sequence.

7.9.1 Modification of Control Architecture

By re-examining the previous control architecture design, it has been determined that the operations of the new section conveyor is similar to the operations of the disassemble conveyor. Additionally, there is no new type conveyor component and new type hardware. Related to this new element, the modification was to change the

interface of the conveying system, which led to the insertion of two items to the original interface namely In5Request and In5OK. In order to achieve the mixed production, the cylinder type needed to be identified. In practice, one proximity sensor was introduced to distinguish 4-cyls CHs from 5-cyls CHs. With the aim of minimising the impact on the architecture and the system, the sensing signal connected to the assembly station that carries out the assembly operation via the system co-ordinator. Therefore, there is neither modification in the assembly station component nor of the associated components.

7.9.2 Modification of Control Component

Modifications were required on both the 'machine system' and 'modular machine' levels. Due to the similarity with the CONV_DEASSEMBLY, the control logic component VEConv1 was re-used with minimal modification (i.e. different inputs and outputs). This kind of re-use also contributed to the time reduction in the development process. In contrast, the system coordinator was nearly re-developed in order to cope with the new AGV operation cycle. It was regarded as the main development in this test case.

7.9.3 Evaluation

In this scenario, the main evaluation point was to measure the impact of the system and the extra development effort that has been added in. A similar measurement method to that used in the first scenario was applied here. Table 7.7 shows the modification percentage of the system.

	Conveyor System (bytes)	Assembly Station (bytes)	System Coordinator (bytes)
Original	70,102	84,626	42,155
Modification	2,454	0	31,706
Percent	3.5%	0%	75.2%

Table 7.7 The Modification of the System

As table 7.7 shows, the modification of the conveyor system and assembly station is minimal. However, the modification of the system co-ordinator is quite high. The main reason for this is the close link between the system co-ordinator and the specific task. In this scenario, the new AGV operation cycle is facilitated by the system co-ordinator, leading to the most modification. From the whole system point-of-view, however, the modification percentage is considerably low.

7.10 Summary

This chapter assesses the effectiveness of the VIR-ENG approach and toolset in system design, implementation and runtime support for agile manufacturing machine systems, through the implementation of a demonstrator cell in Euromation. In particular, the assessment emphasises the aspects of control system development and the use of the CSDE tools via three test scenarios.

The case study of the demonstrator illustrated the VIR-ENG approach and the associated tools in supporting the design and implementation of an operational industrial assembly system. This system is representative of a full production facility involving the cooperative design in multiple disciplines, which can exhibit the agility of the development process. The integration of virtual and real environments in building machine systems has been clearly demonstrated. It is evident that an overall architectural view, provided by the control architecture design, is important in the machine system development process. The creation of the control architecture at an early stage has proven its worth as the basis for the development of virtual and real control components, and runtime support. The concept of developing control logic components in the virtual environment to be used in the real environment has proven to be viable. This process involved development of IEC 1131-3 based control logic in MMDE for test and verification using the simulation model, which was then transferred to CSDE for further development and deployment within the real control system.

Component (particularly software components) reuse is an important feature in the component-based development and it enabled primary aspects of the agility of

machine systems. The main feature of this agility that demonstrated is the capability to cope with change requirements, which was undertaken in the two test scenarios. The consistency and compatibility of component interfaces played an important role in ensuring seamless integration between the virtual and real environments. Overall, product changeover and the introduction of product variants were achieved in the scenarios.

Furthermore, the benefits of reusability are more exploitable when both mechanical hardware and control software adopt the modular design principle (e.g. conveyor modules of the conveying system). The development process also showed the importance of a rich component library, where the components created from the project and during the cell development formed the basis for building a more complete component library. The experience gained reinforces the importance of comprehensive documentation to ensure effective use of the components. Although considerable work is called for in continuing to uncover principles, lessons and techniques to design and implement control systems for agile manufacturing machine systems, the results to date are already useful for immediate application in practical control systems.

Chapter 8 Conclusions and Recommendations

This research study has covered most of the major aspects of the lifecycle of the control system development process for agile machine systems. This chapter provides a summary of the key conclusions and an outline of the potential areas for further research.

8.1 Summary

This research work adopted a component-based system philosophy to facilitate 'agility' in the development process as well as the machine system. Bearing in mind that customer requirements change in an uncertain manner, the component-based system development approach can rapidly adapt to changes by means of limiting the change impact in a certain range of components. The approach has been extended to allow cooperation with other aspects of design during the development process in a flexible manner. The system is built from components. Component-based control systems demonstrate promising agility in control system realisation and contribute to agility in customised machine systems by means of replacing or upgrading components.

The primary research achievements that have extended the body of previous knowledge in the field are as follows:

a) a method to facilitate architecture design classified as conceptual design;
b) a method adopting IEC 1131-3 to implement the conceptual design in a component-based fashion classified as implementation;
c) a prototype set of tools that enable system designers and builders to utilise the concepts, methods, and tools for the use and design of components.

8.1.1 Control Architecture Design

Control architecture design provides a blueprint for the efficient and effective development of the control solution of a machine system. It can also be considered as

a connection mapping to other related design activities. The main research contributions are:

- A method named Component Responsibility and Collaboration (CRC) has been proposed. Rather than introducing a brand-new method to designers, it is intended to encourage the use of a proven mature method within a design framework and some general underlying principles.
- Unified Modelling Language (UML), the standard notation, is adopted as the design description language to represent the control system design. Since it is proposed by the software engineering community and does not fully support control system requirements, some extensions have been introduced to accommodate the necessary control system features.

8.1.2 Control Architecture Design Tools

In order to efficiently construct the control architecture, a supporting tool has been developed. It provides a user-friendly design environment, in which a number of graphic functions have been embedded so that the designer can readily comprehend the tool's functions. It also relieves a lot of the clerical work, and allows the designer to concentrate on the task of requirements analysis and control architecture design.

The main feature of the support tool is that it provides a meaningful graphic structure to organise the elements of the control architecture design. Besides the essential UML notation for the design activities, it also provides a hierarchical tree view by which to organise the generated diagram. Design patterns are embedded in the support tool with a view to encourage reuse of proven designs. Report generation is another feature of the tool.

Furthermore, this tool is an open-end environment. It allows the designers to introduce new methods and/or new notation in order to adapt to different development backgrounds.

8.1.3 IEC 1131-3 and Component-based Design Paradigm

During the detailed design process, IEC 1131-3 has been adopted to facilitate component-based control logic design. The main contributions here are:

- The method transforms the control architecture design into a specific control system design using the Sequential Function Chart (SFC). The logic view is transformed into function blocks and variables; the process view is transformed into a SFC description.
- New concepts have been introduced, in terms of using IEC113-3 languages to develop the control system solution, in order to fit the component-based development paradigm. Based on the control component structure, the IEC1131-3 project and/or resource units are used to represent the individual components. These projects and/or resource units can be simultaneously developed. During the development process, SFC, function block (FB), and structure text (ST) are the recommended languages. After every project and/or resource unit has been validated in the virtual environment, it will be packaged into the component. Finally, these components are assembled into the final control system. During this process, some necessary refinement will take place, in order to meet the specific features of the system.

8.1.4 Underlying Mechanism and Supporting Tools

Because the standard IEC 1131-3 languages do not support component-based development, some underlying mechanisms have necessarily been developed. The functions required for component-based development use existing components and package newly developed artefacts into a new component. Two mechanisms have necessarily been introduced, which are the 'connector' and 'adapter'.

Connector addresses the reuse of existing components. By utilising the standard extension mechanism 'C' function, it establishes the connection with the IEC 1131-3 program. Ports and sinks embedded in the connector establish the connection with the components. As this establishes a bridge to connect the IEC 1131-3 program and the components, it also provides the mechanism to manage the connection and convert

the information. Through a well-defined interface, it allows a user to modify the connection configuration and to dynamically change the connection. The support tool VCon aids the development process.

Adapter addresses packaging issues involved in component-based development. By introducing the component template, it provides a systematic way to package the IEC 1131-3 program into a component. It allows the IEC 1131-3 program to export functions that can be reused by other parts of the system or future systems. CBuilder is introduced to assist the development process. It provides a user-friendly interface to aid the development process and allows the engineer to concentrate on control system development rather than needing to understand the complex underlying mechanism.

8.1.5 Integration with Other Design Environments

Integration with other system development tools is a key issue addressed in chapter 6. The main integration considerations involve the control system design, the mechanical system design and the run-time support system design.

Between the control system design and the mechanical design, integration can be considered in two aspects.

- The control architecture design provides vital information to the mechanical design, so that the mechanical system can be built from virtual components for prototyping the control system and validating the control logic. It is also important for evaluating the whole machine system design.
- IEC 1131-3 programs can be transferred between the two environments. The IEC 1131-3 language is taken as the neutral language, which is used to describe the control logic of the system. The control logic can take advantage of the virtual environment at the prototyping stage by validating the logic without needing the validation of hardware components. Furthermore, it can be capable of transferring back to the virtual environment when the system needs further development without disturbing the real machine system.

Between the control system design and the run-time support system design, the integration can be considered in two aspects.

- CADE (Control Architecture Design Environment) provides the facility to assist the run-time support design. By introducing new notation into the control architecture design environment, the run-time support system design can be carried out within the control architecture design. It provides seamless interaction between two aspects of development.
- The underlying component mechanism allows the run-time support system to gather run-time information for the operator and the wider manufacturing system context. It also ensures the capability to replace or upgrade control components in the run-time system.

8.1.6 Realisation (Case Study Demonstrator Verification)

The complete methods and tool set have been evaluated through implementation of a demonstrator cell, whereas the original concepts were conceived with reference to current assembly machine system requirements at an engine assembly plant. The demonstrator cell implemented at Euromation in Sweden represents a real production facility. It includes typical industrial production machines, such as AGVs, conveyor systems, assembly machines, and gantry loaders. Subsequently, to illustrate the change capability of the machine control system and the capacity of the methods and tools to handle changes, two examples of change were catered for. The results showed the methods and tools can significantly reduce the time to introduce a new system by taking advantage of the component-based control system. The proposed approach can also facilitate machine system modification/reconfiguration.

8.2 Recommendations for Further Work

Some issues that were outside the scope of this research study are identified below and could form the basis for valuable future research:

- *Distributed simulation.* Because of the basic platform constraint, the simulation in the virtual world is carried on a single machine. This does not match the

distributed paradigm of the real system. The testing of the communication between the components is missing. Unpredictable communication among the system components can cause some problems in the real system.

- *Network communication.* Network communication traffic is not considered in the control system design and implementation. This is based on the assumption that the limitation of network transmission capacity will not be reached. However, this may not reflect the real situation and requires further consideration.
- *Remote diagnostics.* Current workbenches only provide a local diagnostic capability. This may not be sufficient for some industrial applications. Remote diagnostics can be implemented using web-based technology relying on the Internet or Intranet to transfer real time information from the machine systems to the other locations for prognosis and diagnostic purposes.

References

Abdallah I. B., Elmaraghy H. A., Elmekkawy T. (2002), Deadlock-free Scheduling in Flexible Manufacturing Systems using Petri nets, *International Journal of Production Research, vol.40, no.12, pp. 2733-2756.*

Albus, J., Barbera, A., and Nagel, N. (1981), Theory and Practice of Hierarchical Control, *Proceedings of the 23rd IEEE Computer Society International Conference, pp. 18-39.*

Allmendinger, G. (1987) Cell Control Evolution, *Six Annual Control Eng. Conf., pp. 80-83.*

Alvers, J. (1988), Personal Computers and Programmable Controllers Coexistence on the Factory Floor, *Seventh Annual Control Eng. Conf., Section XV, pp. 23-34.*

Andy W. and Pete C. (1997), Transaction Integration for Reusable Hard Real-time Components, *IEEE database.*

Anon (1996), Integration of Control System on Agricultural Tractors and Implements, *DHSM LINK Grant report.*

Anon (1997), Integrated Approach to the Design of Control System for High Speed Machines, *DHSM LINK integration project with Aston University.*

Anon (1998), COMPAG (COMponent-based Paradigm for Agile automation) *IMI Grant report.*

Arthanari, J. (2002), Embedded Solutions for Industrial Automation, *Proceedings of the National Conference on Industrial Automation and Intelligent Systems 2002, pp. 74.*

Bailo C. P, Yen C. J. (1997), Open modular architecture controls at GM Powertrain - Technology and Implementation, *SPIE-Int. Soc. Opt. Eng. Proceedings of Spie - the International Society for Optical Engineering, pp. 52-63.*

Balakrishnan, S. and Somasundaram M. (2001), Large Scale Object-Oriented Application Development in Practice, Technology Review #2001-05, Tata Consultancy Services, *http://www.tcs.com/techbytes/htdocs/Large_Scale_OOADfn128_11.pdf.*

Bauer, A., Browne, J., Bowden, R., Duaggan, J. and Lyons, G. (1991), Shop floor Control Systems – from Design to implementation, *Chapman & Hall, New York.*

Bjorke, O. (1979), Computer Aided Part Manufacturing, *Computers In Industry, Vol. 1, No. 1, pp. 3-9.*

Bonfè M. and Fantuzzi C. (2000), Mechatronic Objects Encapsulation in IEC 1131-3 Norm, *Proceedings of the 2000, IEEE International Conference on Control Applications, Conference Proceedings (Cat. No.00CH37162), IEEE, pp. 598-603.*

Booch G (1994), Object Oriented Design with Application, *2nd Edition, CA: Benjamin/Cummings.*

Brennan, R. W., Balasubramanian, S., and Norrie, D. H. (1997), A Dynamic Control Architecture for Metamorphic Control of Advanced Manufacturing Systems, *Proceedings of the International Symposium on Intelligent Systems and Advanced Manufacturing, pp. 213-223.*

Brennan, R. W., Norrie, D. H. (1998), Evaluating the Performance of Alternative Control Architectures for Manufacturing, *Proceedings of IEEE International Symposium on Intelligent Control, pp. 90-95.*

Brennan R. W., Zhang X. K., Xu Y. F., Norrie D. H. (2002), A Reconfigurable Concurrent Function Block Model and its Implementation in Real-time Java, *Integrated Computer-Aided Engineering, vol.9, no.3, pp. 263-279.*

Brooks F. P. Jr. (1987), No Silver Bullet: Essence and Accidents of Software Engineering, *Computer, pp10-19.*

Brown A.W., and Wallnau K.C. (1996), Engineering of Component-based Systems, *Component-based Software Engineering: Selected Papers from the Software Engineering Institute, pp. 7-15.*

Brown A.W. (1996), Preface: Foundations for Component-based Software Engineering, *Component-based Software Engineering: Selected Papers from the Software Engineering Institute, Edited by IEEE Computer Society Press, Los Alamitos, pVII-X.*

Brown A. W. and Wallnau K. C. (1998), The Current State of CBSE, *IEEE Software, vol.15, no.5, pp.37-46.*

Bunce P.G. (1988), Developments in Advanced Factory Control and Manufacturing Systems, Control and programming in Advanced Manufacturing, *IFS Publications Ltd., Springer-Verlag.*

Camus J-L, and Le Sergent T. (2001), Combining SDL with Synchronous Data Flow Modelling for Distributed Control Systems, *SDL 2001: Meeting UML. 10th International SDL Forum, Proceedings pp.1-18.*

Castillo L, Fdez-Olivares J, Gonzalez A. (1999), A Knowledge-based Tool for the Automated Synthesis of Petri nets for Manufacturing Systems, *1999 7th IEEE International Conference on Emerging Technologies and Factory Automation. Proceedings ETFA '99 (Cat. No.99TH8467), IEEE, Vol.1, pp. 643-652.*

Chaar J. K. (1990), A Methodology for Developing Real-time control software for Efficient and Dependable Manufacturing System, *Ph.D. Thesis, University of Michigan.*

Chen, D. (2000), Research Issues on System Architecture for Mechatronics Systems, *http://www.artes.uu.se/events/gsconf00/papers/chen.pdf.*

Chisholm, A. (1998), OPC/OLE for process control overview, *World Batch Forum.*

Cho, H. and Wysk R. A. (1993), A Robust Adaptive scheduler for an Intelligent Workstation Controller, *International Journal of Production Research, vol.31, no.4, pp.771-89.*

Cho, H., Smith, J. S., and Wysk, R. A. (1997), An Intelligent Workstation Controller for Integrated Planning and Scheduling of an FMS Cell, *Production Planning & Control, vol.8, no.6, pp.597-607.*

Cho, H., Jung, N. and Kim, M. (1996), Enabling technologies of agile manufacturing and its related activities in Korea, *Computers and Industrial Engineering, Vol. 30, No. 3, pp.323-334.*

Chryssolouris G., Wright K., Pierce J., Cobb W. (1988), Manufacturing Systems Operation: Dispatch Rules versus Intelligent Control, *Robotics & Computer-Integrated Manufacturing, vol.4, no.3-4, pp. 531-544.*

Chryssolouris G, Pierce J, Dicke K. (1991), An Approach for Allocating Manufacturing Resources to Production Tasks, *Journal of Manufacturing Systems, vol.10, no.5, pp.368-382.*

Clark, K. and Fujimoto, T. (1991), Product Development Performance: Strategy, Organisation and Management in the World Auto Industry, *Harvard Business School Press, Boston, MA.*

Clarke D. W. (2000), Intelligent instrumentation, *Transactions of the Institute of Measurement & Control, pp. 3-27.*

Clements P. C. (1996), Coming attractions in software architecture, *Technical Report CMU/SEI-96-TR-008. Pittburg: Carnegie Mellon University Software Engineering Institute.*

Coad P. and Yourdon E. (1991), Object-Oriented Analysis, *2nd Edition, Englewood Cliff NJ: Yourdon Press/Prentice Hall.*

Cobb, C. K., Gray, W. H., Hewgley, R. E., Klages, E. J., Neal, R.E. (1997), Integrating a Distributed, Agile, Virtual enterprise in the TEAM Program, *SPIE-Int. Soc. Opt. Eng. Proceedings of Spie - the International Society for Optical Engineering, pp. 48-68.*

Colquhoun, G. J., Baines, R. W. and Crossley, R. (1993), A state of the art review of IDEF0, *International Journal of Computer Intergrated Manufacturing, vol.6, no.4, pp252-264.*

ControlShell (1998), Whitepaper, ControlShell: The Component-Based Programming System for Complex Electromechanical System, *Real-Time Innovation, Inc.*

Cooke, R. (2001), Understanding and Performing Reviews of SDLC Project Software Products, *Edpacs, Vol. 28, No. 7, pp.1-9.*

Cox B. J. and Novobiski A. (1991), Object-Oriented Programming – An Evolutionary Approach, *2nd Edition, Reading MA: Addison-Wesley.*

Curto B., Garcia F. J., Moreno V., Gonzalez J., Moreno A. (2001), An Experience of a CORBA-based Architecture for Computer Integrated Manufacturing, *ETFA 2001, 8th*

International Conference on Emerging Technologies and Factory Automation, Proceedings (Cat. No.01TH8597), IEEE, Vol.2, pp. 765-769.

D'Souza, D. F. and Wills, A. C. (1999), Objects, Components and Frameworks with UML: The Catalysis Approach, *Reading MA: Addison-Wesley.*

David, R. and Alla, H. (1992), Petri Nets and Grafcet: Tools for Modelling Discrete Events Systems, *Prentice-Hall.*

De Vin, L. J., Moore, P. R., Pu, J., Steiner, S. J., De Vicq, A. and Medland, A. J. (2002), Approaches to agile manufacturing.

Deen S. M. (1993), Co-operation Issues in Holonic Manufacturing Systems, *Information Intelligent Systems for Manufacturing, pp. 401-412.*

Desrochers, A. A. and Robert Y. A. (1995), Application of Petri nets in Manufacturing Systems: Modeling, Control, and Performance Analysis, *IEEE Press.*

Diedrich C. and Neumann P. (1998), Field Device Integration in DCS Engineering Using a Device Model, *IECON '98, Proceedings of the 24th Annual Conference of the IEEE Industrial Electronics Society IEEE, pp. 164-168.*

Dilts D. M., Boyd N. P., Whorms H. H. (1991), The Evolution of Control Architectures for Automated Manufacturing Systems, *Journal of Manufacturing Systems, vol.10, no.1, pp.79-93.*

Ding J., Zhao J. R., Zhou Y. F., Wang H. F., Wang Y., Ni Y. M., Dou R. H., Xia S. B. (2001), Substation SCADA System Based on Component Technology, *Automation of Electric Power Systems, vol.25, no.18, pp.51-54, 58-59.*

Doiteaux C., Couturier J. F., Richard J., and Bajic E. (1991), Integration and Control Loop of a Flexible Manufacturing Cell, *Computer Applications in Production and Engineering, Proceedings of the Fourth International IFIP TC5 Conference on Computer Applications in Production and Engineering: Integration Aspects, CAPE '91. North-Holland. pp. 401-408.*

Dornier, G. (1990), Evaluation of Standards for Robot Control System Architecture-Final Report Executive Summary, *European Space Agency Contract Report.*

Dornier, G. (1991), Baseline A&R Control Development Methodology Definition Report, *European Space Agency Contract Report.*

Duffie, N. A. (1987), Nonhierarchical Control of a Flexible Manufacturing Cell, *Robotics and Computer-Integrated Manufacturing Systems, pp. 175-179.*

Duffie, N. A., Chitturi, R., Mou, J. (1988), Fault-tolerant Heterarchical Control of Heterogeneous Manufacturing System Entities, *Journal of Manufacturing Systems, pp. 315-327.*

Duffie, N. A. and Prabhu, V. V. (1994), Real-time Distributed Scheduling of Heterarchical Manufacturing Systems, *Journal of Manufacturing Systems, Vol. 13, pp. 94-107.*

Duffie, N. A. and Prabhu, V. V. (1996), Heterarchical control of highly distributed manufacturing system, *International Journal of Computer-integrated Manufacturing, Vol. 9, pp270-281.*

Duran, D. and Batocchio, A. (1994), A High-level Object-Oriented Programmable Controller Programming Interface, *ISIE '94, pp. 226-230.*

Ellsberger J., Hogrefe D. and Sarma A. (1997), SDL - Formal Object-oriented Language for Communicating Systems, *Prentice Hall Press.*

Englander R. (1997), Developing Java Beans. *O'Reilly Press.*

Færgemand O. and Olsen A. (1994), Introduction to SDL-92, *Computer Networks and ISDN Systems, pp. 1143-1167.*

Fantoni D., Chatelet P. (2000), Instrument diagnostics and maintenance via the Internet/fieldbus integration technology, *Emerging Technologies in Measurement and Control, Technology Update LV. Technical Papers of ISA, ISA EXPO 2000, ISA, pp. 137-142.*

Fernandez E. B., Wu J., and Hancock D. R. (2001), A Design Method for Real-time Object-oriented Systems Using Communicating Real-time State Machines, *Managing Information Technology in a Global Environment, 2001 Information Resources Management Association International Conference, Toronto, Ont., Canada, pp. 20-23.*

Fischer C., Olderog E-R, Wehrheim H. (2001), A CSP view on UML-RT structure diagrams, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, Proceedings (Lecture Notes in Computer Science Vol.2029), pp. 91-108.*

Glanzer D. A., Santori M. L. (1998), PC-based control system using fieldbus, *National Manufacturing Week Conference Proceedings '98, 'Preparing Industry for the 21st Century', Reed Exhibition, pp. 61-66.*

Goh A. S. and Moore P. R. (1994), A Mechatronic Solution to a Packaging Application Using an 'Open' Control Integration Platform, *UK/Hungarian International mechatronic Conference, Budapest, Proceedings mechatronic – the basis for new industrial development, Computational Mechanics Publications, pp. 91-96.*

Goldman, S. L., Nagel, R. N. and Preiss, K. (1994), Agile Competitors and Virtual Organizations: Strategies for Enriching the Customer.

Gong, D. C. and Lin, K. F. (1994), Conceptual Design of a Shop Floor Control System from IDEF0, *Computer Ind. Eng., pp. 119-122.*

Gupta, D. and Buzacott, J. A. (1989), A Framework for Understanding Flexibility of Manufacturing Systems, *Journal of Manufacturing Systems, Vol. 8, pp. 89-97.*

Hadj-Alouane N. B., Chaar J. K. and Naylor A. W. (1990), The Design and Implementation of Control Software of the Prismatic Machining Cell, *Cell for Research on Integrated Manufacturing, The University of Michigan, Ann Arbor, MI.*

Hammer, H. (1987), Flexible Manufacturing Cells and Systems with Computer-intelligence, *Robotics & Computer-Integrated Manufacturing, pp. 39-54.*

Harrison, R., Weston, R. H., Moore, P. R. and Thatcher, T. W. (1987), A Study of Application Areas for Modular Robots, *Robotica, pp. 217-221.*

Harrison, R. (1991), A Generalised Approach to Machine Control, PhD Thesis, *Department of Manufacturing Engineering, Loughborough University of Technology.*

Harrison, R., West, A. A. and Wright, C. D. (2000), Integrating machine design and control, *International Journal of Computer Integrated Manufacturing, Vol. 13, No. 6, pp. 498-516.*

Harrison, R. C. (1998), OPC DCOM white paper, *Intellution, Inc.*

Harry, H. C. (1997), Plug-and Play Open Architecture Integration of Mechatronic Systems for Agile Manufacturing, *Open Architecture Control Systems and Standards, Proceedings, pp. 136-145.*

Hatvary, J. (1985), Intelligence and co-operation in heterarchical manufacturing systems, *Robotics & Computer-Integrated Manufacturing, pp. 101-104.*

Hodgson, R. (1991), The X Model: A Process Model for Object-oriented Software Development, *In Proc. Of Toulouse' 91 (Toulouse, France) pp. 713-728.*

Hogg T., Huberman B. A. (1992), Controlling chaos in distributed systems, *IEEE Transactions on Systems, Man & Cybernetics, vol.21, no.6, Nov.-Dec. pp.1325-1332.*

Hoshi T. (1999), Current and future Java technology for manufacturing industry, *IEEE SMC'99 Conference Proceedings, 1999 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No.99CH37028), IEEE, Vol.6, 1999, pp. 404-409.*

IEC (1993), Programmable controller – part 3: programming languages, *International Standard, IEC 61131-3:1993.*

IEC (2000), Programmable controller – part 7: fuzzy control programming, *International Standard, IEC 61131-7:2000.*

Inamoto, A. (1990), The Architecture of Integrated Factory Automation, *IFAC 11th Triennial World Congress, Estonia, USSR, pp.327-332.*

ISA-S95 (2000), ANSI/ISA-95.00.01, Enterprise-Control System Integration, Part 1: Models and Terminology.

ISA-S95 (2001), ANSI/ISA-95.00.02, Enterprise-Control System Integration, Part 2: Object Models Attributes.

ISO, (1990), ISO 9506-1. Industrial Automation Systems – Manufacturing Message Specification – Part 1: Service Definition, *Geneva: International Standard Organization.*

Jacobson I., Christerson M., Jonsson P. and Overgaard G. (1992), Object-Oriented Software Engineering: A Use Case Driven Approach, *Workingham, England: Addison-Wesley.*

Jahnke J. H. (2001), Engineering Component-based Net-centric Systems for Embedded Applications, *Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9), pp. 10-14.*

Jiao J. B. and Yu H. (2001), CORBA-based Distributed Remote Manufacturing System, *High Technology Letters, vol.11, no.10, pp. 88-90.*

Johnson, M. K., James R. (1990), Towards a Distributed Control Architecture for CIM, *Proceedings of CIMCON '90, NIST Special Publication 785, pp. 166-176.*

Johnson T. L., Baker A. D. (1990), Trends in Shop Floor Control: Modularity, Hierarchy, and Decentralization, *5th IEEE International Symposium on Intelligent Control 1990 (Cat. No.90TH0333-5), vol.1, pp. 45-51.*

Jones, A. T., McLean, C. R. (1985), A Production Control Module for AMRF, *Proceedings of the 1985 ASME Computers in Engineering Conference.*

Jones, A. T., McLean, C. R. (1986), A Proposed Hierarchical Control Model for Automated Manufacturing Systems, *Journal of Manufacturing Systems, pp. 15-25.*

Jones, A. T., Barkmeyer, E. J. (1990), Toward a Global Architecture for Computer Integrated Manufacturing, *Proceedings of CIMCON '90, NIST Special Publication 785, pp. 1-20.*

Jones, A. T., Saleh, A. (1992), A Multi-level/Multi-layer Architecture for Intelligent Shopfloor Control, *International Journal of Computer Integrated Manufacturing, pp. 60-70.*

Jones, A., Wallace E. and Yih Y. (2000), Monitor and Controlling Operation, *The Handbook of Industrial Engineering.*

Joshi, S., Wysk R., Jones, A. T. (1990), A Scaleable Architecture for CIM ShopFloor Control, *Proceedings of CIMCON '90, NIST Special Publication 785, pp. 21-34.*

Kang W. J., Park H. S., Lee H. (2001), Design and Performance Evaluation of DCS Using DCOM, CORBA and Tspace, *Proceedings of IFAC Conference on New Technologies for Computer Control, NTCC 2001. Univ. Hong Kong, pp. 54-58.*

Kanitkar V. and Delis A. (2001), Time Constrained Push Strategies in Client-Server Databases, *Distributed & Parallel Databases, vol.9, no.1, pp. 5-38.*

Kavi, K. M. and Yang, S. M. (1989), Real-time System Design Methodologies – an Introduction and Surveys, *Machine Design, February, pp. 539-553.*

Kendricks, L. E. (1987), An Approach to Using Personal Computers and Programmable Controllers for Flexible Cost-Effective Control Systems, *Sixth Annual Control Eng. Conf., Rosemount, IL, May 19-21, pp. 622-626.*

Kerckhoffs E. J. H. and Snorek M. (2001), Analysis of Ventilation Process of a Road Tunnel and its Control System, *Modelling and Simulation 2001, 15th European Simulation Multiconference 2001. ESM'2001. Prague, Czech Republic. ASIM. Arbeitsgemeinschaft Simulation. CASS, Chinese Assoc. Syst. Simulation. et al. pp. 6-9.*

Kim, C., Kim, K. and Choi, I. (1993), An Object-Oriented Information Modelling Methodology for Manufacturing Information Systems, *Computer Industry Engineer, pp. 337-353.*

Koepfer, G. C. (1995). Agile: it's about machines too, *Modern Machine Shop, 67, 10.*

Koren, Y. (1983), Computer Control of Manufacturing Systems, *New York London: McGraw-Hill.*

Koren, Y., Joane, F., Pritschow G. (1998), Open Architecture Control Systems, Summary of Global Activity, *ITA Series, Vol. 2, Milano, Italy.*

Kosanke, K., Vernadat F. B. (1996), CIMOSA: Open Architecture for CIM-An Example for Specification and Statement of Requirements for GERAM, *CIMOSA Association, Version 2.0, Boblingen, Germany.*

Kusiak, A. and He, D. W. (1997), Design for Agile Assembly: an Operational Perspective. *International Journal of Production Research, vol.35, no.1, Jan., pp.157-178.*

Lee, Y. K. and Park, S. J. (1993), Opnets: An Object-oriented High-level Petri nets Model for Real-time System Modeling, *Journal System Software, pp. 69-86.*

Levin V., Basbugoglue O., Bounimova E. and Inan K. (1996), A Bilingual Specification Environment for Software/Hardware Co-design, *Proceedings of the Eleventh International Symposium on Computer and Information Sciences, ISCIS, Middle East Tech. Univ. Part vol.1, pp.153-162.*

Lin J. T. and Ming P. T. (2001), Designing a Holon-based Manufacturing Control System for Automated Storage and Retrieval Systems (AS/RSs), *Journal of Applied Systems Studies, vol.2, no.1, pp. 127-151.*

Liu, C. M. and Wu, F. C. (1993), Using Petri nets to solve FMS problems, *International Journal of Computer Integrated Manufacturing, vol.6, no.3, pp. 175-185.*

Lutz P., Sperling W. (1995), Communication System for Open Control Systems, Opening Productive Partnerships, Concerted Efforts for Europe, *Proceedings of the Conference on Integration in Manufacturing (formerly CIM - Europe Conference), IOS Press, pp. 393-404.*

Lutz P., Sperling W. (1997), OSACA-the Vendor Neutral Control Architecture, Facilitating Deployment of Information and Communications Technologies for Competitive Manufacturing, *Proceedings of the European Conference on Integration in Manufacturing. Tech. Univ. Dresden, pp. 247-256.*

Maarssen L. A. and McGowan C. L. (1981), Structured Analysis and Design Technique (SADT), *Informatie, vol.23, no.7-8, pp.433-442.*

Maffezzoni, C., Ferrarini, L. and Carpanzano, E. (1999), Object-Oriented Models for Advanced Automation Engineering, *Control Engineering Practice, Vol. 7, No. 8, pp. 957-968.*

Maimon, O. Z. (1987), Real-time Operational Control of Flexible Manufacturing System, *Journal of Manufacturing Systems, pp. 125-136.*

Malan, R., Letsinger, R. and Coleman, D. (1996), Object-Oriented Development at Work, *Upper Saddel River NJ: Prentice Hall.*

Manji J. F. (1992), Integrating PCs and PLCs Yields a Formula for Success, *Controls & Systems, vol.39, no.11, Nov. pp.46-48.*

Marotta F., Morzenti A., Mandrioli D. (2001), Modeling and Analyzing Real-time CORBA and Supervision and Control Framework and Applications, *Proceedings 21st International Conference on Distributed Computing Systems, IEEE Comput. Soc., pp. 567-74.*

Martin J. and Odell J. J. (1995), Object-Oriented Methods: A Foundation, *Englewood Cliffs NJ: Prentice Hall.*

Maturana, F. and Norrie, D. (1996), Multi-agent Mediator Architecture for Distributed Manufacturing, *Journal of Intelligent Manufacturing, Vol. 7, pp. 257-270.*

Mayer, R. J., Painter, M. K. and DeWitte, P. S. (1994), IDEF Family of Methods for Concurrent Engineering and Business Reengineering Applications, *Knowledge Based Systems, College Station, TX.*

McKenna F. (1992), Field-bus Overview and Latest Developments, *Proceedings of Field-bus '92 Conference, Institute of Measurement and Control.*

Mehrabi, M. G., Ulsoy, A. G. (1997), State-of-the-Art in Reconfigurable Manufacturing Systems, *Report #2, Vol. I and Vol. II, Engineering Research Center for Reconfigurable Machining Systems (ERC/RMS), The University of Michigan, Ann Arbor, MI.*

Meilir, P. J. (2000), Fundamentals of Object-Oriented Design in UML, *Dorset House, New York.*

Merchant, M. E. (1980), The Factory of the Future-Technological Aspects, *in Towards the Factory of the Future, PED-Vol. 1, American Society of Mechanical Engineers, NY, pp. 71-82.*

Merchant, M. E. (1985), World Trends and Prospects in Manufacturing Technology, *Proceedings of the International Conference on Future Development in Technology the Year 2000, Inderscience Enterprises, pp.261-279.*

Mikell P. G. (2001), Automation, Production Systems, and Computer-integrated Manufacturing, *Prentice-Hall, Inc..*

Moore P. R., Weston R. H., Cooper M. and Armstrong N. (1993), The application of the International Standard Fieldbus Network in the Manufacturing and Process Industries, *DTI Small Scale CIM programme, U.K., Loughborough University and Infa Communications Ltd.*

Moore P. R. and Armstrong N. A. (1994a), A Modular Distributed Architecture for Crane Control, *International Journal of Automation in Construction, pp. 44-53.*

Moore P. R. and Weston R. H. (1994b), Machine-Monitoring and Control of Hoists in Industrial Environments, *BRITE-EURAM BREU-0206.*

Moore, P. R., Chong, S. K., Pu J., Wong, C. B., Steiner, S. J., De Vicq, A. and Medland, A. J. (2002), ARMMS - Agile and Reconfigurable Manufacturing Machinery Systems, *Proceedings of the 8th Mechatronics Forum International Conference, Mechatronics 2002, pp. 24-26.*

Naylor A.W. and Volz R.A. (1987), Design of Integrated Manufacturing Control software, *IEEE Transactions on Systems, Man and Cybernetics, pp. 321-224.*

Niere J. and Zundorf A. (1999), Using Fujaba for the Development of Production Control Systems, *Proceedings of AGTIVE - Applications of Graph Transformations with Industrial Relevance, pp. 1-3.*

Northcott, J. (1988), The Impact of Microelectronics, *Section 4, Forms of Use, PSI Research Report 673, pp 56-65.*

Ohman M., Johansson S., Arzen K. E. (1998), Implementation Aspects of the PLC Standard IEC 1131-3, *Control Engineering Practice, pp. 547-555.*

Olsson, G., Piani, G. (1993), Computer Systems for Automation and Control, Prentice Hall International.

OMG, (1995), The Common Object Request Broker: Architecture and Specification, *Revision 2.0, OMG Document.*

OMG, (1998), CORBA-based Machine Control White Paper, *Version 0.14, OMG Document.*

OPC, (2001), OPC Foundation demonstrates OPC XML and Microsoft .NET at ISA 2001, *Available: http://www.opcfoundation.org/02_news/opc_pr-xml-spec.*

Ozgur U. H., Anlagan O., Engin Kilic S., Cangar T. (2000), A structured methodology for development of heterarchical control software for manufacturing cells using Windows-DNA, *Proceedings of the IASTED International Conference Intelligent Systems and Control, pp. 35-44.*

Paidy S., Reeve R. (1991), Software Architecture for a Cell Controller, *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences (Cat. No.91TH0350-9). IEEE Comput. Soc. Press. vol.2. pp. 339-349.*

Pandy, K.V. (1992), Development of Rules for Production Planning and Control at the Cell level, *Proceeding or the 3rd International Conference on Factory 2000 – Competitive Performance Through Advance Technology, pp. 228-233.*

Peeters J., Jadoul M., Holz E., Wasowski M., Witaszek D. and Delpiroux J. P. (1995), HW/SW co-design and the simulation of a multimedia application, *Proc. 7th European Simulation Symposium.*

Perry, D. E., Wolf, A. L. (1992), Foundations for the Study of Software Architecture, *ACM SIGSOFT, pp. 40-52.*

Perry, S. (1999), Industrial Ethernet: The Death Knell of Fieldbus? *http://www.synergetic.com/education/death_knell_of_fieldbus.pdf.*

Perry, S. (2001), Eight Open Networks and Industrial Ethernet: A Brief Guide to the Pros and Cons for Users and OEMS, *http://www.synergetic.com.*

Petri C. (1962), Kommunikation mit Automaten, *Ph.D. diss., University of Berlin.*

Petri C. (1980), Introduction to General Net Theory, in Net Theory and Applications, W. Brauer (ed.), *Lecture Notes in Computer Science No. 84. NY: Springer-Verlag.*

Philippe, K. (1995), Architecture Blueprints- The "4+1" View Model of Software Architecture, *IEEE Software, pp. 42-50.*

PLCopen (2001), Technical Specification, PLCopen - Technical Committee 2 – Task Force, Function blocks for motion control, *http:// www.plcopen.org/.*

Pritschow, G. (1990), Automation Technology- on the Way to an Open System Architecture, *Robotics and Computer Integrated Manufacturing, Vol. 7, No. 1/2, pp. 103-111.*

Prabhu, V. V. and Duffie, N. A. (1995), Modeling and Analysis of Non-linear Dynamics in Autonomous Heterarchical Manufacturing System Control, *Annals of CIRP, Vol. 44, pp. 425-428.*

Pu, J. and Moore P. R. (1995a), A Comparative Study of Approaches for Building Intelligent Machine Control Systems, *11th National Conference on Production Research, De Montfort University.*

Pu J. and Moore P. R. (1995b), Component/Image-based Design of Distributed Manufacturing machine Control Systems, *ICRAM '95, Istanbul, Turkey.*

Pu, J. and Moore, P. R. (1998), Towards Paradigm Shift in Machine Design and Control, *Proceedings of the 6th UK Mechatronics Forum International Conference, Skövde, Sweden, pp. 23-30.*

Puschner P. and Wellings A. (2001), A Profile for High-integrity Real-time Java Programs, *Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2001, IEEE Comput. , pp. 15-22.*

Pyoun, Y. S. and Choi, B. K. (1994), Quantifying the Flexibility Value in Automated Manufacturing Systems, *Journal of Manufacturing Systems, Vol. 13, pp. 108-118.*

Quinn, R. D., Causy, G. C., Merat, F. L., Sargent, D. M., Barentdt, N. A., Newman, W. S., Velasco, V. B., Podgurski, A., Jo, J. Y., Sterling, L. S. and Kim, Y. (1997), An Agile Manufacturing Workcell Design, *IIE Transaction, pp. 901-909.*

Rana S.P. and Taneja S.K. (1988), A Distributed Architecture for Automated Manufacturing Systems, *International Journal of Advanced Manufacturing Technology, pp. 81-98.*

Raymond B., Eric G. S., André L. and Lionel S. (2001), Enhancing Numerical Controllers, Using MMS Concepts and a CORBA-based Software Bus, *International Journal of Computer Integrated Manufacturing, Vol. 14, No. 6, 560-569.*

Rodd M. G., Dimyai K., Motus L. (1998), The Design and Analysis of Low-cost Real-time Fieldbus Systems, *Control Engineering Practice, pp. 83-91.*

Rogers, P., Brennan, R. (1997), A Simulation Testbed for Comparing the Performance of Alternative Control Architectures, *The Winter Simulation Conference Proceedings, pp. 880-887.*

Roux O. H., Delfieu D., Molinaro P. (2001), Discrete Time Approach of Time Petri nets for Real-time Systems Analysis, *ETFA 2001, 8th International Conference on Emerging Technologies and Factory Automation, Proceedings (Cat. No. 01TH8597), IEEE, Vol. 2, pp.197-204.*

Royce, W. W. (1970), Managing the Development of Large-scale Software System, *Proc. IEEE WESCON, pp. 1-9.*

Rumbaugh J., Blaha M., Premerlani W. ***et al.*** (1991), Object-Oriented Modelling and Design, *Englewood Cliffs NJ: Prentice Hall.*

Schoop R., Neubert R., Colombo A. W. (2001a), A Multiagent-based Distributed Control Platform for Industrial Flexible Production Systems, *IECON'01, 27th Annual Conference of the IEEE Industrial Electronics Society (Cat. No.37243), IEEE, Vol.1, pp. 279-284.*

Schoop R., Neubert R., Suessmann B. (2001b), Flexible Manufacturing Control with PLC, CNC and Software agents, *Proceedings 5th International Symposium on Autonomous Decentralized Systems, IEEE Comput. Soc. 2001, pp. 365-371.*

Selic, B., Gullekson, G. and Ward, P. T. (1994), Real-time Object-Oriented Modelling, *New York: Willey.*

Sessions R. (1997), COM and DCOM: Microsoft's Vision for Distributed Objects, *John Wiley & Sons.*

Seung, H. H. (2000), Experimental performance evaluation of Profibus-FMS, *IEEE Robotics & Automation Magazine, pp. 64-72.*

Shen W. M., Maturana F., Norrie D. H. (2000), MetaMorph II: an agent-based architecture for distributed intelligent design and manufacturing, *Journal of Intelligent Manufacturing, vol.11, no.3, pp.237-251.*

Sheridan, J. H. (1993), Agile Manufacturing: Stepping Beyond Lean Production, *Industry week, 242(8), pp30-46.*

Sinan S. A. (1998), UML in a Nutshell: A Desktop Quick Reference, *'OReilly & Associates, Inc..*

Sklenar J. (2000), Object-oriented Programming in JavaScript, *Proceedings of the 26th ASU Conference Object Oriented Modeling and Simulation, Univ. Malta. pp.35-42.*

Sousa R. M. and Putnik G. D. (1999), Formal Description Technique SDL for Manufacturing Systems Specification and Description, *Proceedings of International Conference on Advances in Production Management Systems 1999: Global Production Management, pp. 6-10.*

Stroustrup B. (1997), The C++ Programming Language, *3rd Edition, Reading MA: Addison-Wesley.*

Struebing, L. (1995), New Approach to Agile manufacturing, *Quality Progress, pp.18-19.*

Szabo, S., Proctor, F. (1997), Validation Results of Specifications for Motion Control Interoperability, *SPIE-Int. Soc. Opt. Eng. Proceedings of Spie - the International Society for Optical Engineering, pp. 166-176.*

Szyperski, C. (1998), Component Software: Beyond Object-Oriented Programming, *Addison-Wesley, Harlow, England.*

Thielemans H, Demeestere L, and Van Brussel H. (1998), HEDRA: Heterogeneous distributed real-time architecture, *Real-Time Systems, vol.14, no.3, pp. 311-323.*

Telelogic (1998), Full Power with SDL and UML, *http://tau.telelogic.com/download/papers/SDLUML.pdf.*

Termini M. J. (1996), The New Manufacturing Engineer: Coming of Age in an Agile Environment, *Dearborn, Mich.: Society of Manufacturing Engineers.*

Ting, J. J. (1990), A Co-operative Shop-Floor Control Model for Computer-Integrated Manufacturing, *Proceedings of CIMCON '90, NIST Special Publication 785, pp. 446-465.*

Tovar, E., Vasques, F. (2001), Distributed computing for the factory-floor: a real-time approach using WorldFIP networks, *Computers in Industry, pp.11-31.*

Underdown, R. and Deese M. (1995), Enterprise Engineering: A Case Study of a Small Manufacturer and Distributor, *Society for Enterprise Engineering Conference Proceedings.*

Upton, D.M., Barash, M.M., Matheson, A.M. (1991), Architectures and Auctions in Manufacturing, *Journal of Computer Integrated Manufacturing, pp. 23-33.*

Valckenaers P., Van Brussel H., Wyns J., Peeters P., Bongaerts L. (1999), Multi-agent Manufacturing Control in Holonic Manufacturing Systems, *Human Systems Management, vol.18, no.3-4, pp.233-43.*

VanBrussel, H., Valckenaers, P., Wyns, J., Bongaerts, L., and Detand J. (1996), Holonic Manufacturing System and IiM, *IT and Manufacturing Partnerships-Delivering the Promise, Edited by J. Browne, IOS Press, pp.188-195.*

Van Brussel H., Wyns J., Valckenaers P., Bongaerts L., Peeters P. (1998), Reference architecture for holonic manufacturing systems: PROSA, *Computers in Industry, vol.37, no.3, pp.255-74.*

Van Hoff, A. (1996), Hooked on Java, *Reading, MA: Addison-Wesley.*

Waller, S. (1985), Strategies for Factory Automation, *Siemens Review, pp.19-23.*

Wallnau, K. (1998), Component Management Infrastructure: Concepts, Trends and Impact, in Kyoto, Japan, *I.W.o.C.-B. Technology, Ed..*

Wang, H. M. and Wang, H. B. (1995), Generation of Optimal Operating Strategies for Robotic Cells: a Petri nets approach, *International Journal of Computer Integrated Manufacturing, vol.8, no.1, pp.32-42.*

Wang L., Balasubramanian S., and Norrie D. (1998), Agent-based Intelligent Control System Design for Real-time Distributed Manufacturing Environments, *Working Notes of Autonomous Agents'98 Workshop on Agent-Based Manufacturing, Minneapolis/St. Paul, MN, May 10, 1998, pp. 152-159.*

Wang, L. H., Sivaram, B. and Douglas, H. N. (1998), Agent-based Intelligent Control System Design for Real-time Distributed Manufacturing Environments, *Agent-based Manufacturing Workshop – Autonomous Agents' 98, May 9-13, pp152-159.*

Warwick, K. (1994), Flexible Distributed Control of Manufacturing System Using Local Operating Networks, *EPSRC/CDP Group, GR/J/64047, University of Reading.*

Wegrzyn M., Adamski M. A., Monteiro J. L. (1998), The Application of Reconfigurable Logic to Controller Design, *Control Engineering Practice, pp. 879-887.*

Weston, R. H. (1989a), Integration Tools Based on OSI Networks, *SME Technical Paper MS89-708.*

Weston, R. H., Harrison, R., Booth, A. H. and Moore, P. R. (1989b), Universal Machine Control System Primitives for Modular Distributed Manipulator System, *International Journal of Production Research, pp. 393-410.*

Weston, R. H., Harrison, R., Booth, A. H. and Moore, P. R. (1989c), A New Approach to Machine Control, *Computer-Aided Engineering Journal, pp. 27-32.*

Weston, R. H., Clement, P., Murgatroyd, I. S. (1994), Information Modelling Method and Tools for Manufacturing Systems, *Procs. Lean/Agile Manufacturing, 94LM094 Aachen, October/November.*

Weston, R. H., Gascoigne, J. D. and Gilders P. J. (1998), Enterprise engineering requirements capture in support of the development of component-based systems, *Input into Joint Meeting of ISO TC 184/SC5/WG1 and the IFAC/IFIP Task Force on Enterprise Integration.*

Weston, R. H. (1998). Integration infrastructure requirements for agile manufacturing systems, *IMechE, pp. 423-437.*

Weston, R. H. (1999), Reconfigurable, Component-based Systems and the Role of Enterprise Engineering Concepts, *Computers in Industry, pp. 321-343.*

Wirfs-Brock R., Wilkerson B. and Wiener L. (1990), Designing Object-Oriented Software, *Englewood Cliffs NJ: Prentice Hall.*

Womack J.P., Jones D.T. and Roos D. (1990), The Machine that Changed the World, *New York: Rawson Associates.*

Wyatt, S. N., Andy, P., Quinn, R. D., Frank, L. M., Michael, S. B., Nick, A. B., Greg, C. C., Erin, L. H., Yoohwan, K., Jayendran, S., and Virgilio, B. V. (2000) Design Lessons for Building Agile Manufacturing Systems, *IEEE Transactions on Robotics and Automation, pp. 228-237.*

Xie J. M., Zhou Z. D., Chen Y. P., Chen B. (2002), Research on the Architecture of Fieldbus-based Open CNC System, *Journal of Huazhong (Central China) University of Science & Technology, vol.30, no.4, pp.1-3.*

Yan L. P., Bao G. F., You J. Y. (2001), Scripting Language for Distributed Components Co-ordination, *Shanghai Jiaotong Daxue Xuebao, pp.188-191.*

Yingst, J. C. (1987), A distributed System Architecture for a Personal Computer to Programmable Controller Interface in Process Control Operations, *Six Annual Control Eng. Conf., Rosemount, IL, May 19-21, pp. 617-621.*

Young, K.W., Muehlhaeusser, R., Piggin, R. S. H., Rachitrangsan, P. (2001), Agile Control Systems for Manufacturing, *Proceedings of the Institution of Mechanical Engineers. Part D, Journal of Automobile Engineering, vol.215, no. D2, pp.189-195.*

Yourdon, E. (1989), Modern Structured Analysis, *NJ: Yourdon Press.*

Zachman, J. A. (1987) A Framework for Information Systems Architecture, *IBM Systems Journal, pp. 276-292.*

Zakarian A., and Kusiak A., (2001), Process Analysis and Reengineering, *Computers & Industrial Engineering, vol.41, no.2, pp. 135-150.*

Zaytoon J. and Carre-Menetrier V. (2001), Synthesis of Control Implementation for Discrete Manufacturing Systems, *International Journal of Production Research, vol.39, no.2, pp.329-45.*

Zha X. F. (2002), A Knowledge Intensive Multi-agent Framework for Cooperative/Collaborative Design Modeling and Decision Support of Assemblies, *Knowledge-Based Systems, vol.15, no.8, pp. 493-506.*

Zielinkski, M. (1999), Fieldbus: a Productivity Enabler, 1999 TAPPI Proceedings, Joint Conference Process Control, *Electrical and Information Conference ISA Pupid 38th Annual Symposium, pp. 279-288.*

Zobel, R. N., and Lee, K. H. (1992), A Graphical User Interface Based Simulation System for Mixed Application Areas, *Proceedings of the 1992 Summer Computer Simulation Conference. Twenty-Fourth Annual Computer Simulation Conference, San Diego, CA, USA, pp. 62-66.*

Appendix A: VCon Manual

A.1 Introduction

The main operations of VCon can be described as:

- Browsing the VECOM component and inside items.
- Organising the specific items that need be exported into ISaGRAF or validated.
- Validating the selected items

A.2 VCon Normal Operation Steps

There are two types of operation associated with VCon. One type refers to the case that not any component has been imported into an IEC 1131-3 program. In contrast, an IEC 1131-3 program may already have been connected with some components, and need to insert new components or re-configure the existed components.

For the first case, the operation process can be described as:

a) Choosing ***Add Component*** from ***Tool***, and then select the correct component.

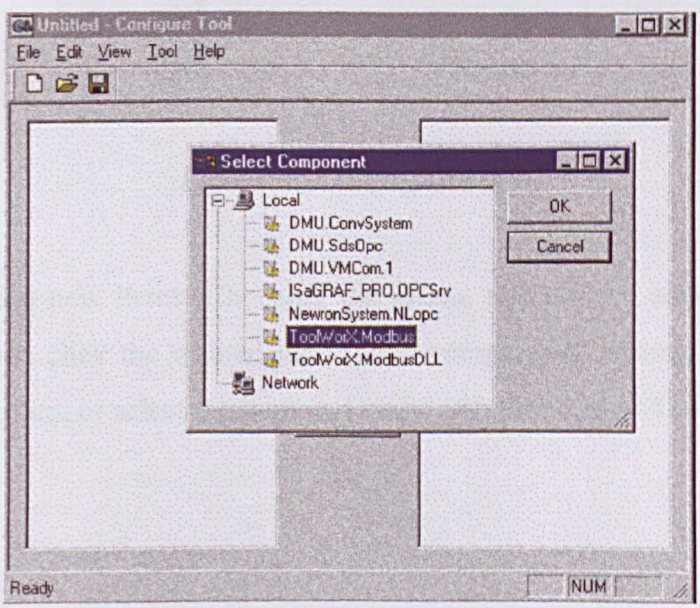

Figure A.1 Selecting Component

b) Generating groups and items, which need to be exported to ISaGRAF or validated.

- Adding group. Choosing ***Add Group*** from ***Tool***. By the default, the group name is *Group1*, and the other parameters are given by the tools. Changing the default group name, deciding update speed, and dead band to the proper ones. The entire items that are composed in the group will be affected by these parameters.

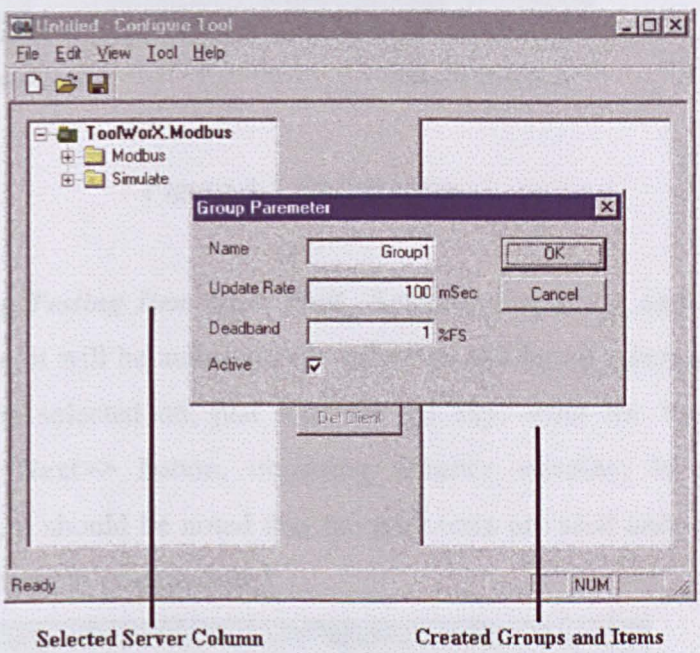

Figure A.2 Adding Groups

- Adding new items. Choosing the group, this will be added in new item. Double click the wanted item (from components column) to be added in the group, or select the item and using add button to add to the group.

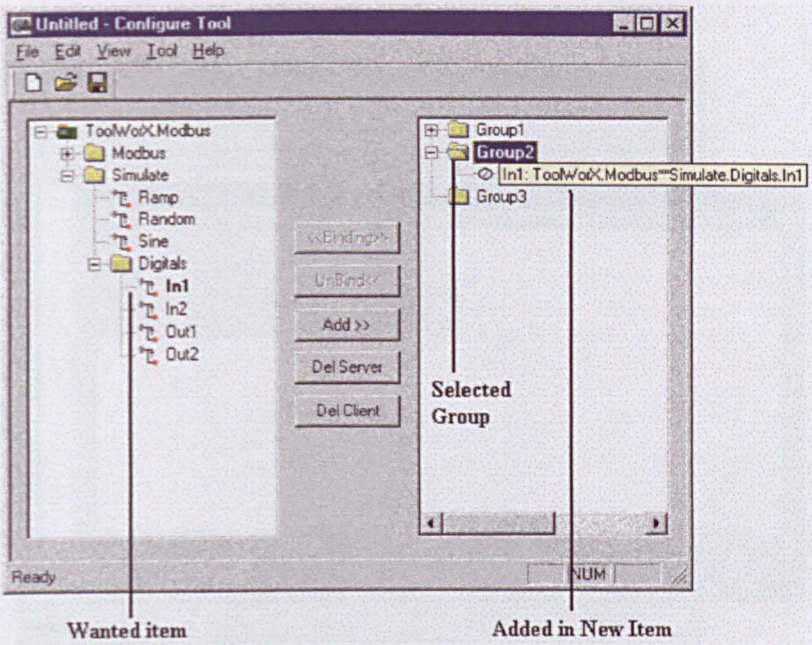

Figure A.3 Adding Items

- Choosing ***Testing Item*** from ***Tool***. Selecting the items, and double click the items, it will be automatically added to validating column. In order to delete the selected on, just uses the del key. After the items is ready, pushing **Next>>** button, obtaining monitor window. In the monitor window, it should be noted that the grey ones are read only items, which means they can't be modified.

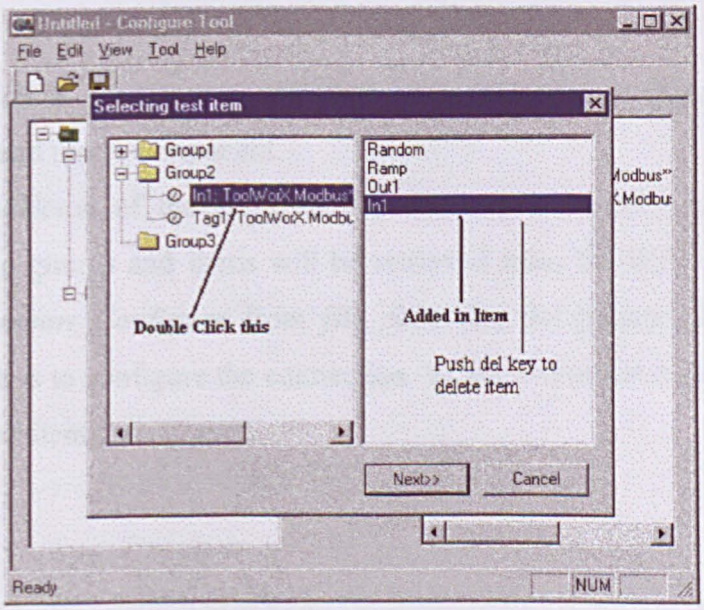

Figure A.4 Testing Items

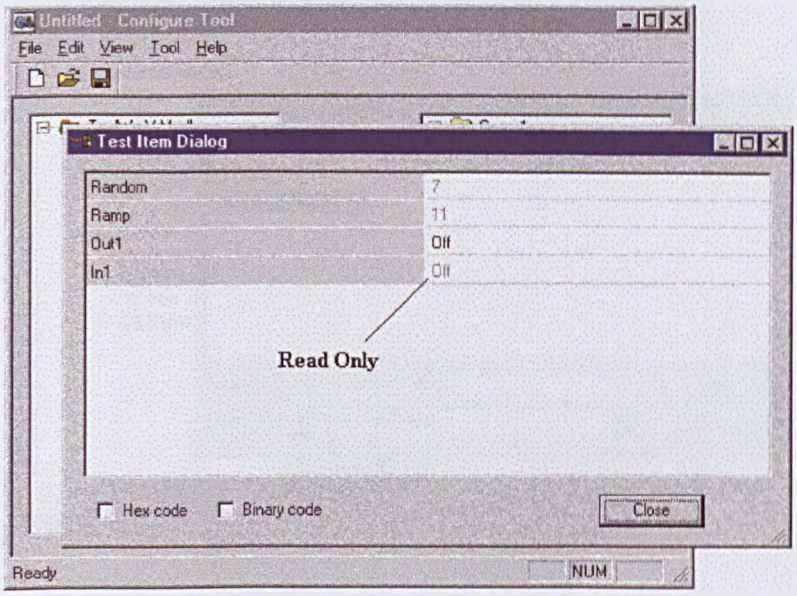

Figure A.5 Monitor Window

- Exporting the configuration. Choosing ***Export Configure*** from ***file***. Selecting ISaGRAF project wanted to export. The configuration for the connector and the IEC-1131-3 code will be generated in the project directory.

For the second case, the most steps are the similar as the first case. The process can be described as:

a) The first step is the same as the first case, which is to discover the proper component and insert component.

b) Since the skeleton of the component connection is already developed in the program, the groups and items will be retrieved from the IEC-1131-3 program. Choosing ***Import Configure*** from ***file***. Selecting the proper ISaGRAF project name, which is to configure the connection. VCon will retrieve the information of the group and items.

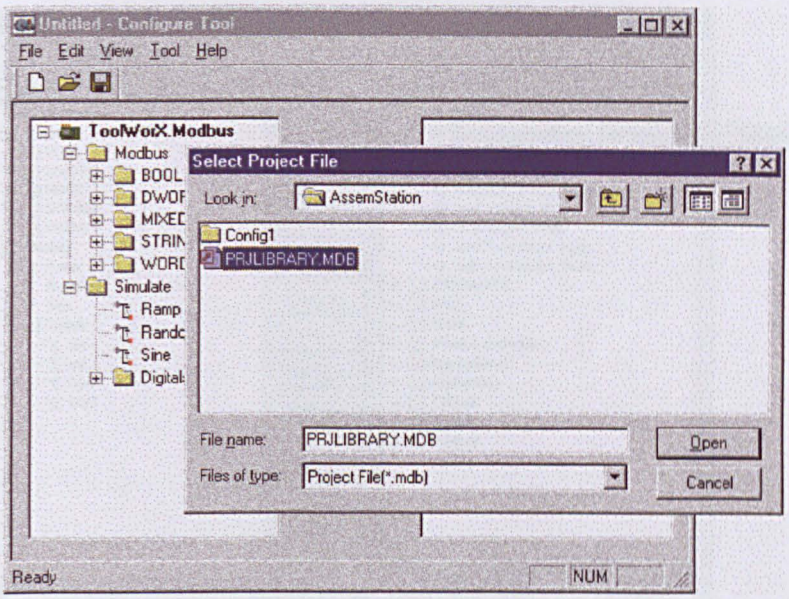

Figure A.6 Selecting Import Project

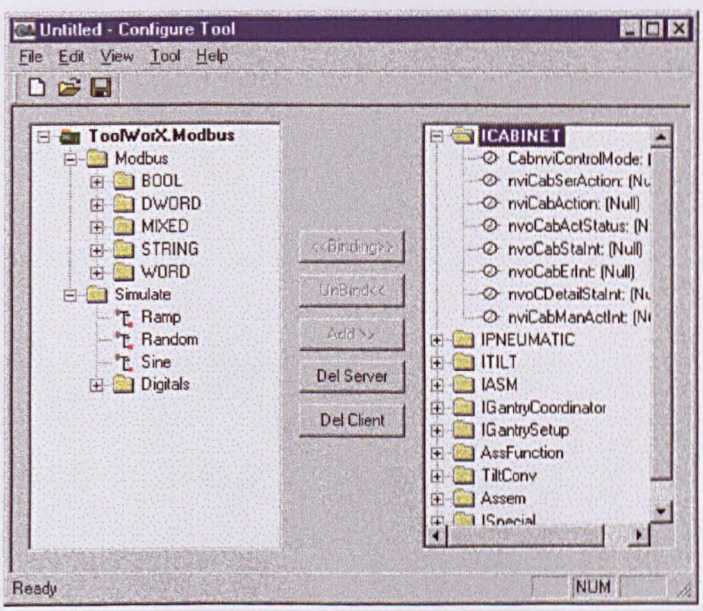

Figure A.7 The Retrieved Groups and Items

c) Building the connection between the component port and the program items. Selecting the component port from the component column, selecting the program items from the generated group items. Click binding button to build the connection. Actually, as the figure shows, the no connection item will show NULL in the following item.

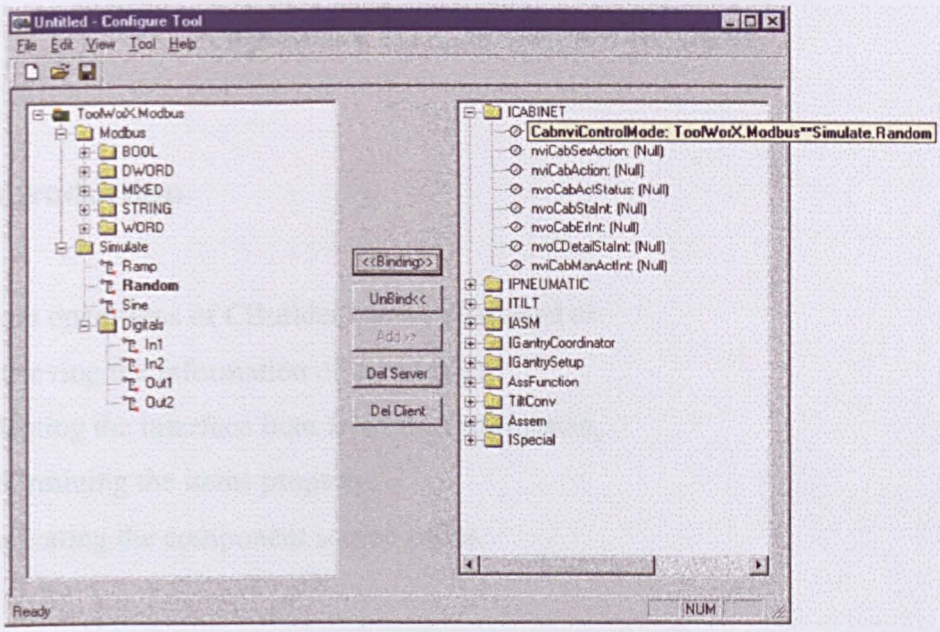

Figure A.8 The Connection Construction

Appendix B: CBuilder Manual

B.1 Introduction

The main operations of CBuilder can be described as:

- Retrieving the information of the project,
- Selecting the interface item from the information,
- Determining the items property,
- Generating the component source codes.

B.2 CBuilder General Operation Steps

The general operation steps of CBuilder can be described as:

a) Selecting the project. Click the right hand side button **Select**, then the project selecting dialog will pop-up. Selecting the proper project that will be built. The information will be retrieved and shown in the left hand side column.

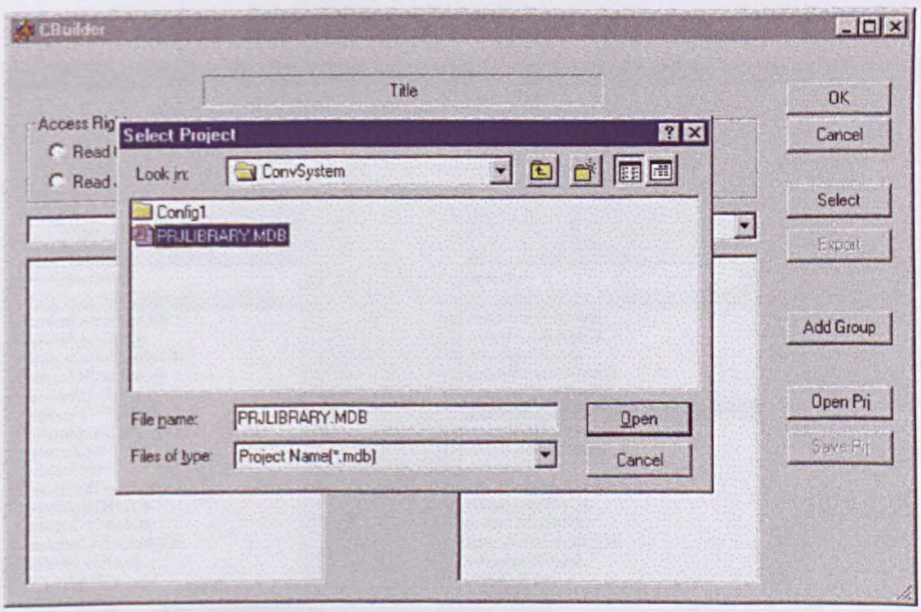

Figure B.1 Importing Project

b) Building the interface. The building interface is separated into two parts. One is to build the group, which is held the items. The new group can be added in by using the left-hand side **Add Group** button. Another is to build the interface items. It can be done by simply selecting the item from the right-hand side, and clicking **Add>>** button. By the default, the access right of items is writable. If the item is read-only, it need change the access right of item by changing the radio button. The whole configuration can be saved in a project for the further modification.

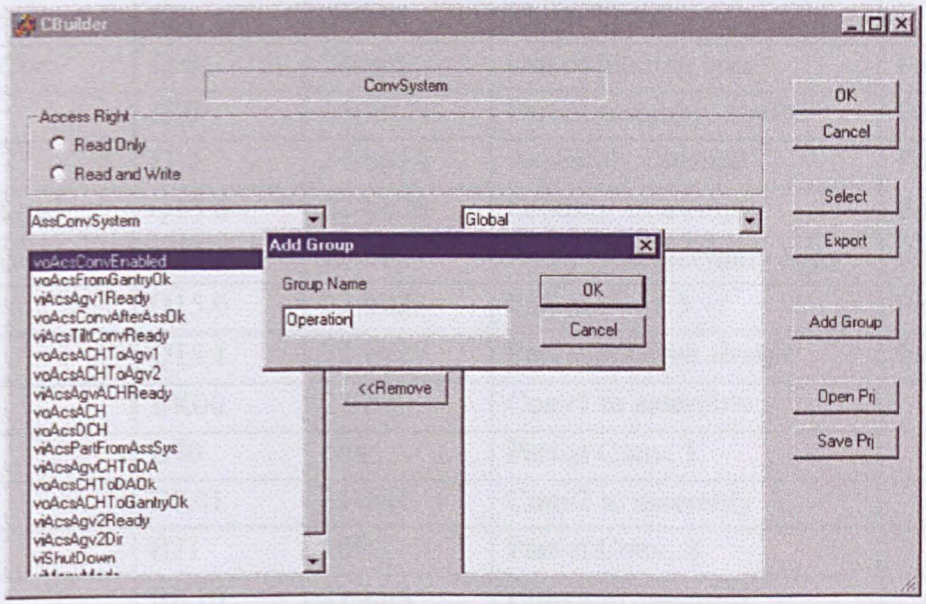

Figure B.2 Adding New Group

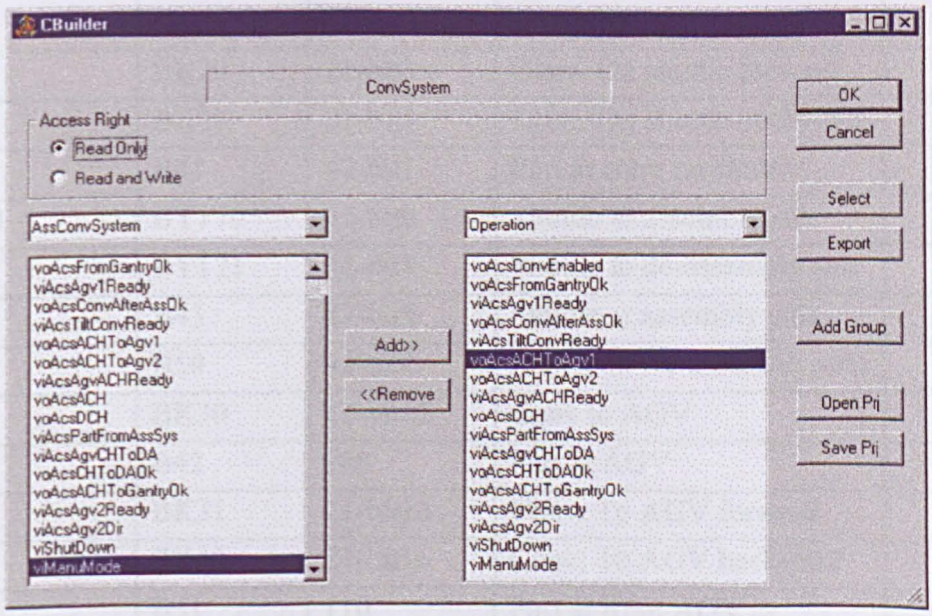

Figure B.3 Adding Items and Adjusting Item Properties

Appendix C: Conveying System Interface List

Variable Name	Instrument Label	SDS Address	Description	Comment
TurnPower	Q12.0	22-bit16	Turn On/Off Signal Power	True for On
PowerErr	I0.0	24-bit0	Fuse Signal	
	I0.2	24-bit2	Instrument Power	
MotorErr	I0.1	24-bit1	Motor Error	
EnableConv	I2.0	24-bit8	Out of loading area	For S7
	I3.0	24-bi12	Part at assembly station	For lon
	I3.2	24-bit14	Assembly finished	For lon
EnableLoad	Q13.0	22-bit20	Loading area empty	For S7
ReadyMove	Q13.1	22-bit21	End-pos deassembly conv	For S7
	Q15.0	22-bit28	Part at Conv 3	For lon
	Q15.1	22-bit29	Part at conv on shuttle	For lon
BK00	BK00	21-bit0	Conv1 to assembly	
B70	B70	104	Part at Conv. 1	
BK01	BK01	21-bit1	Conv2 to assembly	
B71	B71	105	Part at Conv. 2	
BK10	BK10	21-bit4	Conv3 to assembly	
B72	B72	106	Part at Conv. 3	
BK11	BK11	21-bit5	Conv. After assembly	
B40	B40	107	Part at conv. After assembly	
BK20	BK20	21-bit8	Conv. On shuttle forward	
BK21	BK21	21-bit9	Conv. On shuttle backward	
B41	B41	42-bit1	Part at conv on shuttle	
BY170	BY170	41-bit0	Shuttle to assembly side	
BY171	BY171	41-bit1	Shuttle to de-assembly side	
B43	B43	42-bit3	Shuttle at assembly side	
B50	B50	42-bit4	Shuttle at De-assembly side	
BK30	BK30	21-bit12	Conv to AGV	
B42	B42	108	Part at AGV	
BK31	BK31	21-bit13	Conv. To AGV forward	
BK40	BK40	21-bit16	Conv. To AGV backward	
B51	B51	110	Part at from AGV in	
B52	B52	109	Part at from AGV out	

BK41	BK41	21-bit17	Conv De-assembly	
B53	B53	42-bit7	Part at conv De-assembly	
B60	B60	42-bit8	De-assembly Conv. Full	
B80	B80	42-bit16	Part at De-assembly station	
B81	B81	42-bit17	End-pos De-assembly Conveyor	
B82	B82	42-bit18	Part at endstop de-assembly conveyor	
BK50	BK50	21-bit20	Conv To Gantry	
B111	B111	42-bit29	In-pos Conv to Gantry	
B112	B112	42-bit30	End-pos conv to gantry	
B113	B113	42-bit31	Part at endstop conv to gantry	
BK51P	BK51	21-bit21	In-conv. 1 Gantry	
BK51F	Q8.0	22-bit0	In-conv. 1 Gantry forward	
BK51B	Q8.1	22-bit1	Backward	
B90	B90	100	Part at in-conv. 1 Gantry	
BK60P	BK60	21-bit24	In-conv. 2 Gantry	
BK60F	Q9.0	22-bit4	Forward	
BK60B	Q9.1	22-bit5	Backward	
B91	B91	101	Part at in-conv 2 Gantry	
BK61P	BK61	21-bit25	Out-conv. 1 Gantry	
BK61F	Q10.0	22-bit8	Forward	
BK61B	Q10.1	22-bit9	Backward	
B92	B92	102	Part at Out-conv. 1 Gantry	
BK70P	BK70	21-bit28	Out-conv. 2 Gantry	
BK70F	Q11.0	22-bit12	Forward	
BK70B	Q11.1	22-bit13	Backward	
B93	B93	103	Part at out-conv. 2 Gantry	
B100	B100	42-bit24	Stop de-assembly active	
B101	B101	42-bit25	Stop de-assembly deactive	
BY190	BY190	41-bit8	Stop de-assembly activated	
B102	B102	42-bit26	Stop at end of de-assembly active	
B103	B103	42-bit27	Stop at end of de-assembly deactive	
BY200	BY200	41-bit12	End stop activated	

Appendix D: Assembly Station Interface List

Control Component: Cabinet

Component for controlling the main function within the assembly station.

NAME	INTERFACE TYPE	DATA TYPE	DESCRIPTION	NETWORK VARIABLE TYPE	BINDING	REMARK
Icabinet						
nviControlMode	NV input	Int16	1-Automatic mode 2-Manual mode	SNVT_count		
nviCabSerAction	NV input	Int16	0 – No action 1 – Initialise 2 – Emergency stop	SNVT_count		
nviCabAction	NV input	Int16	0 – No action 1 = Magazine pallet change. 2 = Gantry not in magazine area. 3 = Assembly cycle finished.	SNVT_count		Action resets by "No Action"

nvoCabActStatus	NV output	int16	0 – Not initialised 1 – In normal operation 2 – Emergency stop 99 – Error	SNVT_count		Suppose to provide status for automatic operation
nvoCabStatus	NV output	int16	Bit0 = Power is on Bit1 = Magazine ready Bit2 = Magazine pallet in position Others not used	SNVT_state		Suppose to provide status for automatic operation
nvoCabStaInt	NV output	int16	See nvoCabStatus	SNVT_count		
nvoCabError	NV output	int16	Bit0 = Init. Error Bit1 = Internal command error Bit2 = Phase sequence relay not ok Bit3 = Motor overcurrent protection Bit4 = Automatic fuse not ok Bit5 = Automatic fuse mgz	SNVT_state		

			not ok Bit6 = Overtemp in cabinet Bit7 = Magazine level alarm Bit8 = Magazine trouble alarm Others not used			
nvoCabErInt	NV output	int16	See nvoCabError	SNVT_count		
Interface for Manual Operation						
nvoCDetailStatus	NV output	int16	Bit0 = Power is on Bit1 = Magazine ready Bit2 = Magazine pallet in position Bit3 = Part on inconveyor Bit4 = Outconveyor emty Others not used	SNVT_state		Suppose to provide information for maintaining and set-up.
nvoCDetailStaInt	NV output	int16	See nvoCDetailStatus	SNVT_count		
nviCabManAction	NV input	Int16	Bit0 = Power On. (Puls) Bit1 = Bypass limit switches. Bit2 = Magazine pallet	SNVT_state		0 = Off, 1 = On Used only for

			change. Bit3 = Gantry not in magazine area. Bit4 = Assembly cycle finished. Bit5 =Power Off. (Puls) Others not used			manual operation
nviCabManActInt	NV output	int16	See nviCabManAction	SNVT_count		

Table D.1 Cabinet Table

Control Component: Pneumatic

Component for controlling the pneumatic handling

Name	Interface Type	Data Type	Description	Network Variable Type	Binding	Remark
Ipneumatic						
nviControlMode	NV input	Int16	1-Automatic mode 2-Manual mode	SNVT_count		
nviPneSerAction	NV input	Int16	0 – No action	SNVT_count		

			1 – Initialise 2 – Emergency stop			
nviPneAction	NV input	Int16	0 – No action 1 = Main valve on 2 = Main valve off 3 = Air preasure on 4 = Air preasure off 5 = Vaccum on 6 = Vaccum off	SNVT_count		Air preasure is used to push the valves when they have been vacuumed.
nvoPneActStatus	NV output	int16	0 – Not initialised 1 – In normal operation 2 – Emergency stop 99 – Error	SNVT_count		Suppose to provide status for automatic operation
nvoPneError	NV output	int16	Bit0 = Init. Error Bit1 = Internal node error Others not used	SNVT_state		
nvoPneErInt	NV output	int16	See nvoPneError	SNVT_count		
Interface for Manual Operation						
nvoPDetailStatus	NV output	int16	Bit0 = Air preasure is on	SNVT_state		Suppose to

			Bit1 = Vacuum min Bit2 = Vacuum max Others not used			provide information for maintaining and set-up.
nvoPDetailStaInt	NV output	int16	See nvoPDetailStatus	SNVT_count		
nviPneManAction	NV input	Int16	Bit0 = Air preasure Bit1 = Main valve Bit2 = Vacuum Others not used	SNVT_state		0 = Off, 1 = On Used only for manual operation.
nviPneManActInt	NV output	int16	See nviPneManAction	SNVT_count		

Table D.2 Pneumatic Table

Control Component: Tilt Conveyor

Component for controlling the tilt conveyor within the assembly station. Encapsulate the logic of positioning the pallet for assembly.

Name	Interface Type	Data Type	Description	Network Variable Type	Binding	Remark
Itilt						
nviControlMode	NV input	Int16	1-Automatic mode	SNVT_count		

			2-Manual mode			
nviTiltSerAction	NV input	Int16	0 – No action 1 – Initialise 2 – Emergency stop 3 – Stop after complete cycle 4 – Resume	SNVT_count		Change TiltConvAction to TiltServiceAction
nviTiltUAction	NV input	Int16	0 - No action 1 – Tilt to home position 2 – Tilt to inlet position 3 – Tilt to exhaust position	SNVT_count		
nviTiltCvAction	NV input	Int16	1 = Station stop close 2 = Index up 4 = Tilt unit to home position 8 = Tilt unit to inlet position 16 = Tilt unit to exhaust position 32 = Conveyor motor on	SNVT_count		
nvoTiltCvStatus	NV output	int16	0 – Not initialised 1 – Idle	SNVT_count		Suppose to provide status for

			2 – In normal operation 3 – Stopping incoming pallets 4 – Halt (by stop) normal operation 5 – Halt (due to error of other components) exception operation 99 – Error			automatic operation
nvoTiltConvError	NV output	int16	Bit0 = Init. Error bit1 = Internal command error bit2 = bit3 = Station stop error bit4 = Index error bit5 = Tilt error Others not used	SNVT_state		
nvoTiltConvErInt	NV output	int16	See nvoTiltConvError	SNVT_count		
nvoPalletStatus	NV output	int16	Bit0 = No pallet Bit1 =	SNVT_state		Suppose to provide status for

			Bit2 = Bit3 = Pallet at assembly position (State 4 –6used during assembly operation) bit4 = Index up & tilt at home position bit5 = Pallet at inlet position bit6 = Pallet at exhaust position bit7 = Leaving assembly station bit8 = Tilt conveyor error			automatic operation
nvoPalletStaInt	NV output	int16	See nvoPalletStatus	SNVT_count		
nvoPalletType	NV output	int16	0 – No pallet X – Product type no.	SNVT_count		
Interface for Manual Operation						
nvoTDetailStatus	NV output	int16	Bit0 = Pallet at assembly position	SNVT_state		Suppose to provide

			Bit1 = Stop closed Bit2 = Stop open Bit3 = Index down Bit4 = Index up Bit5 = Tilt unit at home position Bit6 = Tilt unit at inlet position Bit7 = Tilt unit at exhaust position Others not used			information for maintaining and set-up.
nvoTDetailStaInt	NV output	int16	See nvoTDetailStatus	SNVT_count		
nviTManualAction	NV input	int16	Bit0 = 0 station stop close; 1 open Bit1 = 0 index down; 1 up Bit2 = Tilt unit to home position Bit3 = Tilt unit to inlet position	SNVT_state		Used only for manual operation

			Bit4 = Tilt unit to exhaust position Bit5 = Conveyor motor Others not used			
nviTManualActInt	NV output	int16	See nviTManualAction	SNVT_count		

Table D.3 Tilt Convey Table

Control Component: Assembly Head

Component for controlling the assembly head together with all its actuators and sensors. Encapsulate the operation logic of the assembly head.

Name	Interface Type	Data Type	Description	Network Variable Type	Binding	Remark
Iasm						
nviAsmServAction	NV input	int16	0 – No action 1 – Initialise 2 – Stop after complete cycle 3 – Emergency stop 4 – Resume	SNVT_count		Change AsmStop to AsmService
nviNumofValve	NV input	Int16	Pick up valve number	SNVT_count		

nviControlMode	NV input	int16	1 – Automatic mode 2 – Manual mode	SNVT_count		
nviAsmHeadAction	NV input	Int16	0 – No action 1 – Pick up valves from magazine with pickers in "right" order e.g. 1,2,3,4 2 – Align and brace valves 3 – Turn unit to assembly position and grab the valves 4 – Insert valves 5 – Check insertion of valves 6 – Pick up valves from magazine with picker in opposite order e.g 4,3,2,1	SNVT_count		
nvoAsmHeadStatus	NV output	Int16	0 – Not initialised 1 – Idle 2 – Operate in automatic cycle 3 – Halt (by stop) normal	SNVT_count		

			operation 4 – Halt (due to error of other components) exception operation 5 – Restart prohibit (e.g. valves in suction cap) 99 – Error			
nvoAsmHeadError	NV output	Int16	bit0 = Init. error bit1 = Internal command error bit2 = Picker 1 error bit3 = Picker 2 error bit4 = Picker 3 error bit5 = Picker 4 error bit6 = Gripper error bit7 = Turning unit error bit8 = Alignment error Others not used	SNVT_state		
nvoAsmHeadErInt	NV output	int16	See nvoAsmHeadError	SNVT_count		

nvoSlave1Output	NV output	Integer	bit0 = Gripper 1+2 release bit1 = Gripper 1+2 brace bit2 = Vacuum 1+2 off bit3 = Vacuum 3+4 off	SNVT_count	Slave 1	To slave 1
nviSlave1Input	NV input	Integer	bit0 = Y-Index bit1 = Y-Limit bit2 = Z-Index bit3 = Z-Limit bit4 = Valve 1 type ok bit5 = Valve 1 type ok bit6 = Valve 1 type ok bit7 = Valve 1 type ok	SNVT_count	Slave 1	From slave 1
nvoSlave2Output	NV output	Integer	bit0 = Picker1 up bit1 = Picker1 down bit2 = Picker2 up bit3 = Picker2 down bit4 = Picker3 up bit5 = Picker3 down bit6 = Picker4 up	SNVT_count	Slave 2	To slave 2

			bit7 = Picker4 down bit8 = Turning unit (assem pos) bit9 = Turning unit (get pos) bit10 = Alignment fwd bit11 = Alignment rev			
nviSlave2Input	NV input	Integer	bit0 = X-Index bit1 = X-Limit bit3 = Z-Collision detector	SNVT_count	Slave 2	From slave 2
Interface for Manual Operation						
nvoADetStatus1	NV output	Int16	bit0 = Picker 1 upper limit bit1 = Picker 1 lower limit bit2 = Picker 2 upper limit bit3 = Picker 2 lower limit bit4 = Picker 3 upper limit bit5 = Picker 3 lower limit bit6 = Picker 4 upper limit bit7 = Picker 4 lower limit bit8 = Gripper 1 released	SNVT_state		

			bit9 = Gripper 1 braced bit10 = Gripper 2 released bit11 = Gripper 2 braced bit12 = Alignment fwd limit bit13 = Alignment rev limit Bit14 = Turning unit Ass position Bit15 = Turning unit Get position			
nvoADetStaInt1	NV output	int16	See nvoADetStatus1	SNVT_count		
nvoADetStatus2	NV output	Int16	Bit0 = Valve in suction cap 1 bit1 = Valve in suction cap 2 bit2 = Valve in suction cap 3 bit3 = Valve in suction cap 4 bit4 = Valve 1 type ok bit5 = Valve 2 type ok bit6 = Valve 3 type ok bit7 = Valve 4 type ok Others not used	SNVT_state		

nvoADetStaInt2	NV output	int16	See nvoADetStatus2	SNVT_count		
nviAManualAction	NV input	Int16	bit0 = 0 Picker 1 - 4 down; = 1 up bit4 = 0 Gripper 1+2 release; = 1 Gripper 1+2 brace bit5 = 0 Turning unit to assembly pos; = 1 Turning unit to get pos bit6 = 0 Alignement rev; = 1 fwd bit7 = 0 Vacuum 1+2 off; = 1 on bit8 = 0 Vacuum 3+4 off; = 1 on Others not used	SNVT_state		Use for system testing and trouble-shooting, maintenance(Manual operation only)
nviAManualActInt	NV output	int16	See nviAManualAction	SNVT_count		

Table D.4 Assemble Head Table

Control Component: Gantry

Component for controlling the gantry (XYZ axes) of the assembly station. Encapsulate the low level Detail of motion control.

Name	Interface Type	Data Type	Description	Network Variable Type	Binding	Node	Remark
IGantryCoordinator							
nviGServAction	NV input	int16	0 – No action 1 – Initialise 2 –Stop with decelerating 3 –Emergency stop 4 – Resume	SNVT_count			
nviControlMode	NV input	int16	1 – Automatic mode 2 – Manual mode	SNVT_count			
nviGantryAction	NV input	Int16	0 – No action 1 – Move 2 – Homing	SNVT_count			
nviAxisXPos	NV input	Real	Axis X absolute position	SNVT_count_f			
nviAxisYPos	NV input	Real	Axis Y absolute position	SNVT_count_f			
nviAxisZPos	NV input	Real	Axis Z absolute position	SNVT_count_f			

nviAxisDemand	NV input	int16	Choose axis for actual position	SNVT_count			1 = X, 2 = Y, 3 = Z
nvoGantryStatus	NV output	Int16	0 – Not initialised 1 – Idle 2 – Moving 3 – Halt (by stop) normal operation 4 – Halt (due to error of other components) exception operation 99 – Error	SNVT_count			
nvoGantryError	NV output	int16	bit0 – Initialisation error bit1 – Internal operation error bit2 – Low-level component error Others not used				
nvoActPos	NV output	structure		UNVT_ActPos			
IGantrySetup							

nviMoveToPos	NV input	real	Move single axis to position	SNVT_count_f			
nviMoveAxis	NV input	int16	Move reference axis	SNVT_count			
nvoHCtrlParam	NV output	structure		UNVT_CtrlParam			These will use structure which was defined in new MCN controller interface. Only different is the uiDeviceID, here 0 is X axis; 1 is Y axis; 2 is Z axis.
nvoHMotionParam	NV output	structure		UNVT_MotionParam			
nvoHUDUnitParam	NV output	structure		UNVT_UDUnitParam			
nvoHIndexParam	NV output	structure		UNVT_IndexParam			
nviHCtrlParam	NV input	structure		UNVT_CtrlParam			
nviHMotionParam	NV input	structure		UNVT_MotionParam			
nviHUDUnitParam	NV input	structure		UNVT_UDUnitParam			
nviHIndexParam	NV input	structure		UNVT_IndexParam			
nviDCAxisNum	NV input	int16	0 – X Axis 1 – Y Axis	SNVT_count			DC means Data collection

			2 – Z Axis				
nviDCAction	NV input	int16	0 – No action 1 – Start 2 – Stop	SNVT_count			
nviDCSampleTime	NV input	int16		SNVT_count			The whole data time range is Sampletime*Range
nviDCRange	NV input	int16		SNVT_count			
nvoDCStatus	NV output	boolean	false – Idle true – Busy	SNVT_switch			
nvoDCCount	NV output	int16		SNVT_count			These are operators to retrieve the stored data. Count gives the useful data number. Index and Value work together to get value one by one
nviDCIndex	NV input	int16		SNVT_count			
nvoDCValue	NV output	structure		UNVT_CollectValue { nvoActPos, nvoDemPos}			

nvoXError	NV output	int16	bit0 = Init error. bit1 = Internal command error. bit2 = Servo amplifier error. bit3 = Following error. bit4 = Encoder counter wrap-around occurred. bit5 = Motor high temperature. bit6 = Long positioning time. bit7 = No index position recorded. bit8 =. bit9 = Ordered position out of range. bit10 = Limit switch activated.	SNVT_state			
nvoYError	NV output	int16	See above	SNVT_state			

nvoZError	NV output	int16	See above	SNVT_state			
IGantryMC							
nvoXServReq	NV output	int16	Service request, X axis Look table MC21 nviServReq1	SNVT_count	nviServReq1	MCN21(a)	
nvoYServReq	NV output	int16	Service request, Y axis See above	SNVT_count	nviServReq2	MCN21(a)	
nvoZServReq	NV output	int16	Service request, Z axis See above	SNVT_count	nviServReq1	MCN21(b)	
nvoCtrlParam	NV output	structure		UNVT_CtrlParam			
nvoMotionParam	NV output	structure		UNVT_MotionParam			
nvoUDUnitParam	NV output	structure		UNVT_UDUnitParam			
nvoIndexParam	NV output	structure		UNVT_IndexParam			
nviCtrlParam	NV input	structure		UNVT_CtrlParam			
nviMotionParam	NV input	structure		UNVT_MotionParam			

nviUDUnitParam	NV input	structure		UNVT_UDUnitParam			
nviIndexParam	NV input	structure		UNVT_IndexParam			
nvoMoveX	NV output	structure	Move position	UNVT_Move			
nvoMoveY	NV output	structure	Move position	UNVT_Move			
nvoMoveZ	NV output	structure	Move position	UNVT_Move			
nviXAlarm	NV input	int16	Alarm X axis Look table MC21 nvoAlarm1	SNVT_state	nvoAlarm1	MCN21(a)	
nviYAlarm	NV input	int16	Alarm Y axis See above	SNVT_state	nvoAlarm2	MCN21(a)	
nviZAlarm	NV input	int16	Alarm Z axis See above	SNVT_state	nvoAlarm1	MCN21(b)	
nviActPos	NV input	structure		UNVT_ActPos			
nviXState	NV input	int16	state axis X	SNVT_state	nvoState1	MCN21 (a)	
nviYState	NV input	int16	state axis Y	SNVT_state	nvoState2	MCN21 (a)	
nviZState	NV input	int16	state axis Y	SNVT_state	nvoState1	MCN21 (b)	

Table D.5 Gantry Table